An Investigation into the Application of Genetic Programming Techniques to Signal Analysis and Feature Detection

Christopher Harris

A dissertation submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy of the University of London

Department of Computer Science
University College London

26 September 1997
Abstract

This thesis investigates the use of the Genetic Programming (GP) machine learning paradigm to computer vision problem domains related to the detection of features within discrete data signals. The use of GP to find a solution to such problems involves two pieces of design. The first is the set of elements that the search process uses to construct possible problem solutions, known as the primitive set. The second is the design of an appropriate reward mechanism for candidate solutions, known as the fitness function. The wide range of signal types and features to be detected in computer vision detection problems ensures that representation of both data and solutions is of prime importance.

Two contributions made by this thesis are to provide approaches to both primitive set design and fitness function design that are generic to feature detection problems. A three-component fitness function design is presented which reflects the desirable properties of a generic feature detector. Work is also presented on a design for a primitive set that is applicable to a wide range of signal processing problems. These techniques are explored using the classic problems of edge detection and template matching as experimental test-beds.

Work in this thesis on edge detection produces filter functions that outperform those produced by human experts under real-world conditions. This process can be used to produce edge detectors optimised for specific sets of data.

Working from the basis that Strongly Typed Genetic Programming (STGP) is essential for solving vision problems, a fourth contribution is the adoption of STGP as a general syntactic constraint mechanism for the production of GP program trees. This provides a structuring mechanism in addition to allowing the use of complex and relevant data types within candidate programs. By using the structuring to enforce a hierarchy of abstraction in terms of data types and representations, we can increase the power of GP to solve hard problems, and allow more intelligent use of the limited search power available with finite computational resources. This is a form of abstraction not previously used in GP.
Acknowledgements

This research was carried out under the Postgraduate Training Partnership Scheme at University College London, in conjunction with SIRA Ltd.

I would like to thank my supervisor, Bernard Buxton, for providing much useful general scientific advice and as a walking bibliography for computer vision material. Bibliographic honourable mentions must also go to Bill Langdon, whose tireless work on maintaining the GP bibliography saved me many hours of fruitless search. Thanks also to Bill for his advice and discussions on matters relating to GP, and for the time he spent proof-reading and commenting early drafts of this thesis. Danny Alexander gave constructive advice of the thesis from the point of view of a non-GPer.

Most of all, I want to thank Catherine Swales, who has made the past three years of my life the happiest and most illuminating ones I’ve ever had. I have learnt from her more than I thought it was possible not to know.
Contents

1. Introduction......................................................................................................12
  1.1 Motivation.................................................................................................13
  1.2 Goals of the thesis ....................................................................................14
  1.3 Structure of the thesis ..............................................................................15
  1.4 Contributions............................................................................................16

2. A Detailed Description of Genetic Programming........................................18
  2.1 Genetic algorithms and genetic programming .........................................18
    2.1.1 Genetic algorithms and genetic search ..............................................18
    2.1.2 Genetic programming ........................................................................22
  2.2 Tackling a problem with GP ....................................................................23
  2.3 Program tree construction methods .........................................................26
  2.4 Convergence.............................................................................................27
    2.4.1 Program tree manipulation .................................................................29
      2.4.1.1 Mutation .......................................................................................29
      2.4.1.2 Crossover ........................................................................................30
      2.4.1.3 Soft brood selection ......................................................................31
      2.4.1.4 Other genetic operators .................................................................31
    2.4.2 Selection methods and convergence ..................................................31
2.9.5 Iteration, recursion and the halting problem .............................................................65
2.9.6 Implementation restrictions - tree sizes, growth, and parsimony ..........................65

3. GP for Detection Problems ...............................................................................................67
   3.1 The nature of detection problems .............................................................................67
   3.2 Desirable characteristics of a detector .................................................................68
   3.3 Designing fitness functions for detection problems .................................................70
      3.3.1 Fitness function complexity and its relation to the problem, to the type system
           and to the chances of success ...............................................................................70
      3.3.1.1 The exploitative nature of GP and fitness function design ............................70
      3.3.1.2 Fitness function types, complexity and STGP ..............................................71
   3.4 Machine learning of detectors with Genetic Programming ......................................72

4. Evolving an Edge Detection filter ..................................................................................75
   4.1 Convolution Filters for edge detection .................................................................76
      4.1.1 The optimal filter. A theoretical treatment of edge detection filters ...............76
      4.1.2 Translation to the digital domain .................................................................78
   4.2 Evolving a filter with Genetic Programming ...........................................................79
      4.2.1 Five preparatory steps ....................................................................................79
      4.2.2 Experimental design .....................................................................................81
      4.2.3 Evaluating the performance of the evolved detectors .....................................84
   4.3 Results and performance evaluation of detectors .....................................................86
      4.3.1 Operator groups .............................................................................................86
      4.3.2 Performance evaluation ..................................................................................86
      4.3.3 Results ............................................................................................................87
         4.3.3.1 Results for underwater test data ...............................................................87
         4.3.3.2 Results for office environment test data ...............................................87
         4.3.3.3 Results for synthetic step edge data ......................................................89
   4.4 Discussion ...............................................................................................................89
   4.5 Evolved edge detection operators ..........................................................................90

5. Template Matching ........................................................................................................96
7.4.3 Office environment signals ................................................................. 149
7.4.4 Underwater environment signals ....................................................... 150

7.5 Discussion .............................................................................................. 152

8. Conclusions ............................................................................................. 154

8.1 The future of GP ....................................................................................... 156
8.2 Future work ............................................................................................. 157

9. Bibliography .............................................................................................. 159

Appendix A: A practical tool for Genetic Programming ................................. 169

A.1 Introduction ............................................................................................ 169
A.2 Structure of a GP system ......................................................................... 169
  A.2.1 Problem-dependent components ....................................................... 171
  A.2.2 Problem-independent components .................................................... 172
A.3 The GP-COM software system ............................................................... 172
  A.3.1 System structure ................................................................................ 173
  A.3.2 The population manager ................................................................... 175
  A.3.3 Code generation ................................................................................ 177
  A.3.4 Parallelism in Genetic Programming ................................................ 181
  A.3.5 Tree generation .................................................................................. 184
  A.3.6 Storage of genetic material ............................................................... 185
  A.3.7 Run-time customisation .................................................................... 185
A.4 Discussion ................................................................................................ 187
List of Figures

Figure 2.1. Effect of the genetic operators in the standard GA ...............................................20
Figure 2.2. The genetic search cycle .............................................................................................21
Figure 2.3. Effect of genetic operators in GP ..............................................................................24
Figure 4.1. A signal containing step edges ..................................................................................77
Figure 4.2. Four analytically derived edge detector functions .................................................80
Figure 4.3. Sampling a filter at different rates ..............................................................................81
Figure 4.4. Example training data ..................................................................................................85
Figure 4.5. Edge detector performances on underwater test data .............................................92
Figure 4.6. Edge detector performances on office environment test data ................................ 92
Figure 4.7. Edge detector performances on synthetic step edge data .......................................93
Figure 4.8 Operator evolved on training data consisting of synthetic step edges corrupted   
with Gaussian noise ..................................................................................................................93
Figure 4.9. Operator evolved on training data taken from underwater image ensemble ....94
Figure 4.10. Operator evolved on training data taken from office environment image ensemble ..............................................................................................................................95
Figure 5.1. Training images used for template matching experiment ......................................102
Figure 5.2. Fitness profiles for the 'face' picture ......................................................................106
Figure 5.3. Fitness profiles for the 'cross' picture ....................................................................107
Figure 5.4. Best of run individuals for 'cross' picture ..............................................................108
Figure 5.5. Best of run individuals for 'face' picture .................................................................108
Figure 6.1. Forming a backbone of connectives with chaining ...............................................116
Figure 6.2. Fitness profiles for line-based and component-based representations on the   
'cross' picture ...........................................................................................................................123
Figure 6.3. Fitness profiles for line-based and component-based representations on the   
'face' picture ..............................................................................................................................124
Figure 6.4. Fitness profiles for component-based representation with chaining ..........125
Figure 6.5. Fitness profiles for component-based representation with transformation isolation ......................................................... 125
Figure 6.6. Fitness profiles for component-based representation with constant constraint .......................................................... 127
Figure 6.7. Best of run individuals for component set (face picture) ........................................................................ 130
Figure 6.8. Best of run individuals for chaining set (face picture) ..................................................................................... 130
Figure 6.9. Best of run individuals for transformation isolation set (face picture) ........................................... 131
Figure 6.10. Best of run individuals for constant constraint set (face picture) ..................................................... 131
Figure 6.11. Structural complexity of best evolved individuals with the structuring methods .................................................................................................................. 135
Figure 6.12. Structural complexity versus fitness for component-based representations ........................................... 135
Figure 6.13. Effect of adding chaining constraint ........................................................................................................ 136
Figure 6.14. Effect of adding transformation isolation ........................................................................................................ 137
Figure 6.15. Effect of adding constant constraint ........................................................................................................ 138
Figure 7.1. Edge detection results for synthetic step edge training signals, no noise ...................................................................... 147
Figure 7.2. Edge detection results for synthetic edges training set ........................................................................... 148
Figure 7.3. Edge detection results for synthetic edges test set ...................................................................................... 148
Figure 7.4. Edge detection results for office environment training set ....................................................................... 150
Figure 7.5. Edge detection results for office environment test set ...................................................................................... 150
Figure 7.6. Edge detection results for underwater environment training set ........................................................................ 151
Figure 7.7. Edge detection results for underwater environment test set ........................................................................ 151
Figure A.1. Conceptual structure of the GP cycle ........................................................................................................ 171
Figure A.2. Top-level structure of GP-COM ........................................................................................................ 176
Figure A.3. Parallelisation methods in GP-COM ........................................................................................................ 182
Figure A.4. GP-COM component locations ........................................................................................................ 184
List of Tables

Table 4.1. The set of terminals used by GP.................................................................82  
Table 4.2. The set of functions used by the GP............................................................82  
Table 4.3. Tableau for edge detection filter problem ..................................................83  
Table 4.4. Theoretical operators used in performance evaluation.................................87  
Table 4.5. Edge detector performances on underwater test data....................................88  
Table 4.6. Edge detector performances on office environment test data.........................88  
Table 4.7. Edge detector performances on synthetic step edge data.............................90  
Table 5.1. Primitive set for template matching problem..............................................100  
Table 5.2. Tableau for the template matching problem................................................101  
Table 5.3. Fitness rewards and penalties for template matching....................................102  
Table 6.1. Component-based primitive set.................................................................114  
Table 6.2. Primitive sets for chaining.........................................................................117  
Table 6.3. Primitive set additions/replacements for transformation isolation...............117  
Table 6.4. Primitive set additions/replacements for constant calculation constraint ......119  
Table 6.5. Summarised results of structuring experiment suite (maximum possible score is 132).......................................................................................................................128  
Table 6.6. Correlations between fitness and structural complexity (number of line segments) .........................................................................................................................134  
Table 7.1. Primitive set for signal-based edge detection..............................................142  
Table 7.2. Tableau for the complete edge detection problem.........................................144
Chapter 1

Introduction

In recent years the Genetic Programming (GP) paradigm of machine learning, pioneered by John Koza, has attracted attention as a promising method of solving problems by computer. A wide variety of problem domains have been shown to be amenable to treatment with GP due to the enormous flexibility of representation afforded. It is an attractive answer to the question of how to get computers to solve problems because it uses the same methods as we do to solve problems on a computer i.e. it creates computer programs.

Applying GP to the task of solving problems of interest to real-world researchers is an important way to see how powerful GP can be. Initially GP was used on small, relatively easy problems, either as a proof-of-concept or as a demonstration that the representations were flexible enough to solve a wide range of problems. [Koza, 1992] shows that GP can solve seemingly very different problems, such as evolving decision trees for input classification and evolving a playing strategy for a ‘Pac-Man’ game, if such problems can be rephrased as problems of program induction.

This thesis explores ways in which GP can be used to solve machine vision problems, in particular detection problems in digital signals. We want to find out to what extent there is a generic understanding of some signal processing or machine vision problems that can be of benefit when using genetic search. Such problems, being rooted in real-world data and troubled by noise and the enormous heterogeneity of inputs, offer a challenge to any machine learning technique that seeks to produce answers without depending on human high-level reasoning. The promise of GP is that the tremendous
power and flexibility of the representations used could offer solutions not available with other learning paradigms. The work in this thesis looks at the special consideration that must be given by the designer of a GP solution space to the issues of primitive set (the elements used by GP to construct solutions to problems) and fitness function (the method of rewarding and penalising solutions by rating their performance on specific tasks) design when tackling detection problems.

1.1 Motivation

While GP is a promising and exciting way to implement a problem solving strategy, at the time of writing many of the problems successfully tackled with GP looked quite simple from the representational point of view. Lots of human effort was devoted to designing an encoding of the problems’ inputs such that the problem domains became very similar when GP was applied. The problems often boiled down to that of evolving a correct number for a given input string, often with very simple fitness functions, and having inputs with much reduced dimensionality due to pre-processing of raw input data e.g. [Tackett, 1993]. This approach limits the classes of problems that can be solved, and the usefulness of GP, by constraining the role GP has to play in producing answers. The use of enhanced representational schemes such as Strongly Typed GP (STGP) [Montana, 1995], in which program trees can return and manipulate arbitrary data types, was minimal and it appeared that problems solved were chosen because they worked well within the framework that vanilla GP (i.e. without typing mechanisms) offered. It was thought important that GP should be applied to problems that had real-world applicability and that would be hard for GP to solve. The problem domains addressed in this thesis, edge detection and template matching, were chosen in part for these reasons.

Another criticism of GP is that it often fails as a problem solving technique on higher-order instances of problems for which lower-order instances are easily solvable [Gathercole & Ross, 1997]. The ability of GP to solve a problem depends to some extent on the computational resources available but also on other techniques used to increase the power of the search process that make more effective use of those resources. The ability of GP to solve high-order instances of problems is known as scalability (see section 2.9.2). This thesis addresses the issue of scalability through the application of type systems to influence the way solutions are constructed with GP (see Chapter 6).
Certain classes of problem cannot be adequately tackled by GP without extending the scope of the technique beyond that of expressions involving simple numbers. Real-world problems are often like this, which is the reason why most computer programs have some kind of data abstraction within them. It was an aim of this work to apply GP to real-world, relevant problem domains, which implies the use of strong typing, hence much of the work in this thesis involves the use of data abstraction through STGP.

Looking at ways in which GP can solve problems with strong typing brings up its own set of questions. Fitness function design becomes orders of magnitude more complex when values produced by programs are not just numbers. Primitive set design becomes a hard design issue since as well as the transformations carried out within the primitives it is now necessary to consider how the trees can (and cannot) be constructed with the constraints of a type system. This thesis considers both of these design issues within the context of detection problems in digital signals. Detection problems are ideal as test-beds for GP for several reasons:

- The multi-dimensional nature of detection problems, varied types of data within signals, and application of statistical methods to those signals all point strongly toward the need to represent many different types of data within program trees. These problems offer tough representational challenges that GP will have to cope with.

- Many detection problems are optimisation problems with inherent noise, where no ‘best’ answer exists, and are thus suitable for solving in a multitude of different ways. This is suited to the application of GP as an alternative to human intuition.

1.2 Goals of the thesis

This thesis aims to develop techniques to improve the performance of GP on tough, real-world detection problems. This means using raw input data that has not been pre-processed, and using GP to detect relevant features within that data. Firstly, we need to show that GP is a technique capable of addressing tough optimisation problems in detection, and producing high quality solutions to such problems. Once this base-level credibility of GP as a technique for solving detection problems is established, the two fundamental issues of reward (fitness function) and representation (primitive set) need to
be addressed and methods for design and implementation of these elements of the machine learning process developed, within the context of computer vision detection problems. Moreover, solution forms should be developed from primitives that have wide applicability, to a whole class of problems, to deflect charges of over-specificity in primitive sets that would make solving problems trivial (almost solving the problem by hand by using extremely powerful problem-specific primitives that dictate solution forms beforehand). Generic primitive sets need improved search capability to work successfully [O'Reilly, 1995, Chapter 3], so we will need to increase the scalability of GP, i.e. the ability of GP to solve instances of problems that it could not solve before.

The goals have to a large extent been achieved. Work on producing edge detection filters in Chapter 4 shows that GP can successfully produce high-quality solutions to a problem where peak performance differences can be only fractions of a percent. The ability of GP to solve binary template-matching problems, though the careful structuring of primitive set type systems, is shown in Chapter 5 and improved upon in Chapter 6. These methods are then applied to the problem of edge detection in Chapter 7 where GP is required to produce a much more complex solution than before, using a primitive set that is generic and could be applied to a range of detection problems.

1.3 Structure of the thesis

After this introduction Chapter 2 presents a detailed description of the process of applying GP to a problem domain. It highlights many of the issues surrounding GP, arising either from its basis in Genetic Algorithms (GA) [Holland, 1992] or from characteristics of GP that are not found in a GA, and briefly describes approaches taken to address those issues. Additionally, Chapter 2 contains discussions of research relevant to this thesis in the application of GP to image processing and signal processing problems.

Chapter 3 discusses the nature of detection problems in general and how we can use the generic properties of detectors as an (implicit or explicit) basis for designing fitness functions. It presents a generic approach to the design of fitness functions for all detection problems, taken from the work of Canny [Canny, 1986] which is explored in Chapter 4.

Chapter 4 presents practical work on the evolution of edge detectors with GP. Using the canonical form of GP as described in [Koza, 1992], this work draws heavily from the computer vision literature to inspire the experimental design. It demonstrates
that GP is capable of producing extremely high-quality solutions to optimisation problems in computer vision.

Chapter 5 presents practical work in applying GP to a template matching problem, using strong typing and a primitive set to draw arbitrary binary templates. This work is also used as a basis for extended template matching experiments in Chapter 6, which demonstrates the use of STGP as a versatile and powerful constraint mechanism for program tree generation. This mechanism is explicitly employed to promote the hierarchical use of different levels of representation within program trees, and to provide methods of structuring that significantly improve performance in the chosen problem domain.

Chapter 7 employs the approaches to both fitness function design and primitive set design developed in Chapters 4, 5 and 6. Practical work on adopting problem-specific representations and conventions in this chapter employs traditional image-processing techniques, applied via GP, to the problem of edge detection. STGP is used to allow GP to produce solutions in the language of the image processing engineer by employing a primitive set that implements elementary techniques used in vision applications.

Chapter 8 concludes the main body of the thesis with a discussion of the methods developed and the results produced by the body of practical work. Conclusions for the application of GP to detection problems in discrete signals are made. Avenues of future work are also discussed.

The software system developed to perform the experiments for the practical work in this thesis is described in Appendix A. This system is used as an example to highlight and discuss issues in software design for genetic programming systems. Particular attention is paid to the structure of the GP cycle and how software can be constructed to reflect that structure.

1.4 Contributions

This thesis makes five main contributions:

1. This thesis considerably extends the amount of work done on applying GP to image processing and computer vision problems so far. It is the first thesis to be devoted to this issue.

2. GP is used to evolve edge detection filter functions that outperform those
designed by human experts on real-world and synthetic data. This work provides new levels of performance for edge detector filters in computer vision and as such is a contribution to that field. Moreover, GP is shown to be able to produce high-quality solutions to problems using raw input data that has not been pre-processed. This gives GP extra credibility as a problem solving strategy in real-world domains where such data is prevalent, and reduces the role of the human in the problem-solving process.

3. Strong typing is presented as a method of adding explicit syntactic constraints to influence the construction of program trees. This technique is extremely powerful and can be used in many ways. This thesis uses such constraints in three ways in Chapter 6 to address problems in the design of a primitive set.

4. Using these constraints, a technique is developed to increase the scalability of GP to solve higher-order instances of some problems by enforcing a hierarchy of abstraction in data types within program trees. Design of primitive sets through judicious use of strong typing permits the GP practitioner to guide the way potential solutions are constructed and increases the ability of GP to find good solutions.

5. GP solves problems using primitives that are relevant to human engineers in the same problem domain. Use of signal-based primitives allows solutions to be analysed post-hoc and increases the relevance of GP-based solutions to engineers and designers. Such primitive sets can be widely applicable to other problems in the same area and, with the combination of improved scalability, allow GP to become a more general problem-solving tool.
Chapter 2

A Detailed Description of Genetic Programming

This chapter of the thesis performs a dual function. Firstly it describes the Genetic Programming process in some detail, going through each stage of solving a problem with GP. Secondly it highlights particular issues that arise when applying GP to a problem, and summarises current research into improved methods that address these issues. For a comprehensive survey of GP see [Langdon, 1996a, Appendix B]. Finally, relevant literature, involving the applications of GP to signal-oriented problems, is briefly surveyed.

2.1 Genetic algorithms and genetic programming

GP draws heavily from its roots in genetic algorithms and many issues in GP are in fact issues in GAs. A description of GP therefore necessarily must describe GAs as well. Where GP comes into its own, and moves beyond conventional GAs, is with the representational flexibility it provides.

2.1.1 Genetic algorithms and genetic search

Both GP and GAs seek to provide solutions to problems by an iterative process of rating and rewarding elements within a population of candidate solutions. Genetic search is a stochastic search process within a space of possible solutions, consisting of a number of discrete stages. The search method iteratively manipulates a finite sized sample of the solution space (the population of potential solutions) in such a way as to increase the chances of finding better-performing solutions. The search is driven by a stochastic process, combining elements which seek to exploit promising areas of the search space
with those that devote resources to speculative exploration of less promising areas of that space. Genetic search is an abstract search method into which problem-specific elements must be added. Solutions are encoded in some form and the range of such encodings defines the size and shape of the solution space. Assessment of these encoded solutions is also problem-specific, and rates the individuals within the population to direct the search process. The stages involved in genetic search are as follows:

1. An initial population of randomly created individuals is created. In the most basic form of GA these are bit-strings encoding parameters to a solution form. Each individual represents a possible solution to the problem, i.e. a single point within the space of all solutions the can be generated with the chosen encoding.

2. Each individual within the population of candidate solutions is rated according to how well it solves the required task. This is done by a problem-specific component called the fitness function, which maps points in the solution space to fitness values, usually single numbers, by assessing the candidate solutions over a number of ‘fitness cases’ (the training set of data), representative of the whole problem domain being tackled.

3. A new population is derived from the old one using the fitness information from stage 2, via the two processes of selection and genetic operation. Selection produces a new population by repeatedly selecting individuals from the old population to be copied into the new. The process is designed to favour high-quality solutions (that is, ones with higher fitness values) by biasing the makeup of the new population. Fitter individuals in the old population get reproduced into the new population more often than less-fit individuals, so a particularly fit individual can appear several times in the new population. This process produces a clustering effect at those already sampled points in the solution space that have higher fitness associated with them. Once the new population has been selected, novel solutions are produced by applying genetic operators to a proportion of the candidates in the new population. Each genetic operator (there may be several different ones) has an associated frequency which states the proportion of the population to which the operator is applied. Genetic operators work by taking a candidate solution and making some kind of
Mutation in GA. Replaces random substring

Crossover in GA. Swaps random substrings

Figure 2.1. Effect of the genetic operators in the standard GA

alteration to the candidate’s genetic makeup (its genome). The most common operators are mutation and crossover, although others can be used (see section 2.4.1.4). Mutation changes a randomly selected portion of the candidate’s genome into a new, randomly generated, piece of genetic code. Crossover swaps randomly selected pieces of genome between two different individuals in the population (see Figure 2.1). The new population may either be constructed in its entirety via the selection methods, and the genetic operations subsequently applied ‘in place’ to the reproduced individuals, or the reproduced individuals may be operated on while being reproduced, before being placed in the new population.

4. The new population takes the place of the old one, and the process repeats from step 2. The cycle stops after some termination criterion has been satisfied, which can be related to the performance of the search or can be a simple upper limit on the number of cycles performed.

Figure 2.2 illustrates this cycle, which applies to both genetic algorithms and to GP. The cycle is a simplified analogy of the Darwinian processes of natural selection and survival of the fittest, where organisms that are better at the ‘problem’ of surviving get to reproduce more than those that fail the test. Population-level evolution comes about when this natural selection works on a diverse range of organisms each of which has different
Figure 2.2. The genetic search cycle

characteristics. The exploitation of these differences leads to the selection of some forms of organism over others, whilst a background of low-rate copying error (mutation) in the organisms' genomes, and the mechanisms of sexual reproduction (crossover) ensures the maintenance of some kind of diversity by producing novel genetic material.

The encoding of the genetic material of an individual is known as the genotype of that individual. This is distinct from the phenotype of the individual, which is the visible expression of the code within the genotype, the effect the genotype has on the outside world. In biology, our genotype consists of strands of DNA, but our phenotype encompasses not only how we look but how we behave and interact with the outside world. In most GA or GP problems the phenotype of an individual is a property of the fitness values it produces.

The classic GA [Holland, 1992] uses bitstrings to represent candidate solutions. These solutions can be encodings of a set of parameters values, to be ‘plugged in’ to a generic solution form which needs optimising, or they can represent the desired solutions directly. For a bitstring of length $n$, there are therefore $2^n$ possible solutions. This solution space can conveniently be thought of as a (many dimensional) landscape, where coordinates of all but one dimension represent the values in the bitstring and are mapped by
the fitness function onto a ‘height’ which represents the quality of the solution. The genetic algorithm is a method of exploring this landscape in such a way that the chances of finding the highest point, the global optimum in the space, are increased.

For many problems, a solution of sufficient quality may not be found before the maximum number of generations (cycles) has been reached. Often it is necessary to repeat the search process several times to find a solution, each time starting with a different initial population (and hence a different sample of the solution space). Each instance of the process is referred to as a *run*.

If we keep the landscape metaphor, the selection process can be seen as a clustering method. After selection, more instances of the better individuals survive and some of the worse individuals no longer appear. This has the effect of increasing the average fitness of the population due to the increase in frequency of individuals with higher fitnesses. The best fitness value in the population remains unchanged by selection, however, since no novel individuals have been created. The subsequent role of the genetic operators is therefore to move individuals around the landscape by altering their makeup, increasing the diversity in the population and allowing the search process to sample new areas of the space. The greater number of instances of fitter individuals means that more novel individuals will be made from copies of relatively fit solutions than from unfit ones. If the solution space, the form of which depends on the chosen encoding of solutions and on the fitness function, has the right characteristics a small permutation of a fit individual stands a chance of being fitter still than the original candidate solution from which it came, leading to the discovery of progressively better and better solutions.

### 2.1.2 Genetic programming

There are three major differences between GP and the classical GA. Whilst GP uses the genetic search process described in section 2.1.1 to evolve the population of candidate solutions, its representations differ enormously from those of the standard GA.

Where the standard GA used strings to represent solutions, the forms evolved by GP are tree-structures. Where the standard GA uses a fixed-length representation, GP trees can vary in length (usually up to some prescribed maximum size or depth). Where standard GAs use a binary alphabet to form the bitstrings, GP uses an alphabet of varying size and content depending on the problem domain. GP trees are made up of internal nodes and leaf nodes, drawn from a set of primitive elements that are specific to the
problem domain being tackled and that the designer thinks may be useful in contributing to a good solution.

The name “Genetic Programming” comes from the notion that we are able to represent computer programs with a tree-structured genome. Computer programming languages can be represented by formal grammars which are tree-based, so it is actually very simple to represent program code directly as trees. Leaf nodes in the program trees are thus known as terminals, and internal nodes as functions. Program trees are randomly created by selecting elements at random from the provided primitives (known as the function set and terminal set, or collectively as the primitive set). When a function is selected for insertion into the tree, we must then select elements to fill the subtrees that represent that function’s parameters.

With the change in representation from GA to GP comes a change in the dynamics of the search process. The search process is governed, against a background of random change, by the interaction of the selection methods, the genetic operators, the fitness function and the primitive set. The selection methods used in GP are the same as those used in GAs but the operators must be redefined to cope with the different structure of the genome. The operators of mutation and crossover are defined to work in a similar way to those used with bitstrings, but on randomly selected subtrees within the genome rather than substrings. This can be seen in Figure 2.3.

2.2 Tackling a problem with GP

Much preparation is required when tackling a problem with GP. The cycle described in section 2.1.1 takes a number of parameters, for which values need to be decided. The problem representation needs to be defined, the encoding of solutions specified and the fitness function designed. Additionally, the following questions need to be answered before experimental runs can take place:

- What is the set of terminals to be used in the program trees?
- What is the set of functions to be used in the program trees?
- What is the fitness measure?
- What values to give the parameters and variables for controlling the run e.g. population size, number of generations?
Figure 2.3. Effect of genetic operators in GP

- How do we know when we have a good enough result?
- When do we terminate a run?
- How many runs do we perform?
- How are program trees to be constructed? What differences exist for trees created by mutation, if mutation is being used?
- What selection method is going to be used?
- What genetic operators, and at what frequencies, are going to be applied?

Each of these factors affects the search process, and consequently the chances of finding a successful solution. The interaction of genomic representation and fitness function together define the nature of the search space. Such interactions are usually extremely complex, and the space is extremely high-dimensional. The selection methods and genetic operators together define the balance between exploration and exploitation, that is, they determine how much effort is devoted to exploring parts of the space that show promise and how much effort goes into looking for new parts of the space that may ultimately be more rewarding.

There may also be practical considerations to think of. Run-times are usually an important factor, so some degree of parallelism in execution might be beneficial. Memory overheads and run-time are also affected by the size of the trees being processed, so some
controls might be needed to limit the maximum size of a program tree. Implementation of
the GP system is also an issue, depending on the problem domain. For instance, the
overhead of the GP system and its operations is usually insignificant when compared to
the evaluation of individuals’ fitness, so choice of representation and execution method of
the candidate programs could be an important one if the problem domain is inherently
computationally expensive.

As an example of what decisions the GP practitioner might make, we will use an
eexample problem that Koza tackles with GP in [Koza, 1992, section 7.3], which is a
simple symbolic regression problem. Given a set of 20 data points randomly sampled from
a curve over the interval [-1.0, 1.0], GP needs to evolve a symbolic expression that passes
through those data points in order to discover the underlying model to that data. Such an
expression would be a functional form which would depend only on the independent
variable x within that interval. Koza thus picks the terminal set as simply

\[ T = \{x\} \]

and supplies a number of arithmetic and trigonometric functions for the function set
(where % is a protected division function, such that dividing by 0 returns 0).

\[ F = \{+,-,\times,\div,\sin,\cos,\exp,\text{rlog}\} \]

Fitness is measured by testing the evolved expressions at the known values of x and
measuring some error value, which is chosen as the absolute value of the error at each
point. These are summed over the 20 cases to produce a total error measure for the
particular program tree. Here a high fitness is indicated by a low error value. Population
size is set at 500, generations at 51 (initial population plus 50 iterations of the cycle). The
run is set to terminate when an individual matches all 20 points to within a tolerance of
0.01 per point. Koza fixes a 90% frequency for the crossover operator and does not use
mutation at all. Selection is fitness-proportionate, meaning that the number of copies of an
individual made in the next generation is directly proportional to the fitness scores
achieved by the individual. Trees are generated using the ‘ramped half and half’ method
(see section 2.3) with a maximum depth of 17 and a maximum depth of 6 for the initial
generation. All genetic operations that produce trees with depths greater than 17 will fail
to produce a new individual and will result in copying the parent(s) unchanged into the
new population.

25
The decisions made are normally tabulated to produce a ‘tableau’, which provides a quick summary of all these run parameters. Tableaux are used in the experimental work detailed in the later chapters of this thesis (e.g. see Table 4.3). The rest of this chapter includes material on the choices available to the designer during each of the steps listed above, most of which seek to improve search performance by reducing the tendency of populations to converge prematurely (see section 2.4). In addition, the methods used in the practical sections of this thesis will be stated.

2.3 Program tree construction methods

Program trees in GP, either as a part of the initial population in a run or by the mutation operator, are generated randomly using elements of the function set and terminal set only to produce tree nodes. Many methods can be used to generate these trees, starting from the root and recursively filling in the lower levels of the tree. In this section we only consider the canonical case of un-typed, un-constrained GP. Enhancements such as strong typing (see section 2.6.3) have their own implications for tree generation (see section 2.9.6).

Koza uses two methods for generating program trees in [Koza, 1992]. The first method, called “full” produces a tree of maximum possible depth, where all leaves in the tree are at the specified maximum tree depth. This is achieved by simply picking from the function set only, for all depths except the maximum depth, where only the terminal set is used. The second method, called “grow” involves picking randomly from the full primitive set at each point in the tree, terminating that branch if a terminal is selected. The depth constraint is again enforced by only picking from the terminal set if the depth limit is reached on a particular branch, but functions or terminals may be chosen at levels higher than the maximum depth. This method produces variably shaped trees, in contrast to the fixed shape trees produced by the “full” method.

In experiments, Koza combines these two methods and calls the resulting method “ramped half-and-half”. This takes a maximum depth value, and generates trees in batches, sliding the depth constraint for each batch from 2 to the maximum depth. For each of these batches, half the set of trees is created using “full” and half using “grow”. This creates a good variability in tree shape and size. In a set of experiments with four sample problems, Koza demonstrated that the ramp technique was usually better than either the full or grow technique [Koza, 1992, section 25.1].

26
For the experiments in this thesis, a different method is used. Instead of constraining the maximum size of the trees by imposing a hard depth limit, a probabilistic method is used to restrict tree growth. This method is explained fully in Appendix A.

2.4 Convergence

In nature, the process of evolution is a continuous one. Species continue to adapt to changing environmental conditions, including competition from other species that are themselves evolving, and do not simply come to a halt as far as evolutionary progress is concerned. Such a halt would be fatal to the species as it then succumbs to increasingly strong competition. Yet in evolutionary computation such stagnation occurs very often, and is called convergence.

Convergence is an issue at the heart of many techniques to improve the performance of genetic search, many of which will be detailed in the rest of this section. The term is used to describe the state of affairs when the population becomes largely homogenous, consisting largely of variations on a single style of solution. This genetic homogeneity comes about when a particular form of solution benefits from increased selection over a number of generations but doesn’t lead to necessarily fitter offspring, and hence dominates the population. Such a phenomenon doesn’t often occur in nature because the environmental pressures are always changing, always dynamic, and the environment itself is incredibly complex. Any species which failed to adapt to its environment would be at a competitive disadvantage and would eventually die out. In genetic search as used in GAs and GP, the environment is rarely dynamic and the search process can often get stuck in a particular area of the solution space.

Convergence in itself is not undesirable; it normally indicates that the search process has found a profitable area and is concentrating on that part of the solution space. The problem is with premature convergence, where the search stops prematurely and doesn’t find a solution of sufficient quality. In seeking to delay convergence, researchers are trying to get the search to last longer before it gets attracted to a single region of the solution space, to increase the chances of finding a better attractor, whilst retaining the tendency to exploit promising regions once found. With finite resources these two desirable properties contend with each other.

Two opposing forces are at work during the genetic search. The first is the selection process, which increases chances for exploitation of good areas of the search
space by increasing the relative frequency of the better solutions in the previous
generation. The second force, which tends to promote exploration of new areas of the
search space, is the action of the genetic operators upon the program trees. These are
responsible for the variation in the population once selection has been completed.

Convergence comes about when a good genotype arises (that is, one with high
fitness relative to the rest of the population) that is phenotypically resistant to changes in
genotype induced by the genetic operators. Once a good solution arises, it is reproduced
several times in the next generation. Leaving aside the chances of other, better solutions
arising from other parts of the population, we can see that the frequency of a highly fit
genotype in a population increases rapidly if instances of that genotype are not perturbed
in some significant way by the action of a genetic operator.

There are three effects a genetic operation can have upon an individual program in
terms of its fitness:

1. The program’s fitness can be relatively unaffected, either because the
operation fails for some reason (a constraint laid down by the designer,
perhaps), or because the operation succeeds but the change in genome has
little effect when expressed as fitness. In this case there is little change from
the previous generation, and the frequency of descendants of that program will
tend to increase rapidly within the population.

2. The operation can have a deleterious effect on the program, reducing its
fitness compared to other instances of the original program in the new
population. In this case the new genotype is likely to be removed from the
population over time, and the increase in instances of the fit genotype is
slowed only marginally by the loss of a single instance. The frequency of the fit
genotype still increases exponentially.

3. The final possibility is that the action of the operator increases the fitness of
the individual. In this case the new fitter genotype will begin to proliferate
throughout the population, at the expense of other, less fit genotypes.

Convergence happens when the first two cases occur much more often than the
third case. When fitter individuals are unlikely to be produced, the system carries on
producing (phenotypically identical or similar) copies of the fittest individuals currently
present, which are already common in the population. The result is a kind of stagnation where the search is increasingly concentrated in a single area of the search space, from which it is increasingly unlikely to escape. This usually leads to a sub-optimal solution (unless the population happens to converge on a globally optimal area), and results in wasted effort on the part of the designer and CPU. The ability of the system to continually produce fitter individuals is referred to as evolvability [Wagner & Altenberg, 1994]. The maintenance of the evolvability of a population is therefore synonymous with the avoidance of convergence, since the likelihood of continually finding fitter individuals (and thus avoiding domination by a single genotype and its phenotypically similar variants) remains high.

Much effort has been expended in modifying the search process to maintain evolvability. The rest of this section concentrates on both novel and traditional techniques which enhance the ability of genetic search to counter the onset of premature convergence. These can be broken down into methods that work at the individual level to increase the probability of producing a fitter individual with a genetic operator, and ones that work at the population level, to reduce the rate of propagation of fitter individuals. Often such methods are combined, and each are at the mercy of the noise present in genetic search. A fit individual may well be lost in the disruption caused by genetic operations before it has a chance to propagate, which is undesirable. This is an unavoidable possibility, however, if we are to stick to a search process with finite storage and computational resources.

2.4.1 Program tree manipulation

Program trees are manipulated by genetic operators. These operators can be unary or binary, and may produce a single offspring or a pair. These operators are responsible for introducing and maintaining the variability in fitness that drives the selection process, and can be seen as working against the selection methods in terms of population diversity.

2.4.1.1 Mutation

Mutation is often used as a generic term for any change to a genome based on random factors. It was not initially widely used in GP for historical reasons, since the original set of GP experiments in [Koza, 1992] did not use it as Koza wished to demonstrate that random search was not involved in GP’s success at solving a wide range of problems. It has often been adopted by other researchers since then due to its role in the maintenance
of diversity. The use of the term mutation comes from the biological metaphor of a random change being induced in a genome as a fault in the reproductive process.

The classical mutation as used in GP is a random replacement of part of a program tree with a newly-created subtree. A point in the program tree for replacement is chosen by a (either uniform or biased) random process, and the subtree headed by this point is excised. A new tree is randomly generated and inserted in its place. This new tree is often subject to depth restrictions in a similar way to the process detailed earlier for whole program trees, but with a much smaller depth limit. A typical depth limit used is 4. Mutation produces a single offspring from a single parent.

Mutation introduces (usually) entirely new genetic material into the gene pool. Although the nodes themselves (which are often called genes despite being a less than accurate analogy) are not new, the likelihood being that many instances of each are already present in the population, the combination of these nodes is quite likely to be unique and thus some novelty is introduced, increasing the overall diversity of the population. It is for this reason that mutation is often used to reduce the chances of premature convergence.

2.4.1.2 Crossover

Crossover is the analogue of the genetic recombination process that takes place between paired chromosomes of DNA in cell meiosis (the process by which reproductive cells, e.g. ova and spermatozoa, are produced in many living organisms). In genetic search it involves the mixing of genetic material from two parent individuals to produce one or two new individuals with different genotypes. A point is chosen at random in the first parent. A point is also chosen at random in the genome of the second parent. The subtrees headed by these nodes are then swapped. Each parent loses a section of its own genome and has it replaced with a section of the other parent’s.

Crossover can be seen as a special case of the plain mutation operator, the essential difference being that the new material introduced into the parent is drawn randomly from the gene pool rather than being generated randomly. Whether this is beneficial is still an open question. It certainly allows the propagation of blocks of code that were of use elsewhere, and is the primary vector for the re-use of code that is likely to benefit fitness. The fact that these blocks have already been used, however, tends to encourage a uniformity in building solutions and can contribute to convergence problems.
2.4.1.3 Soft brood selection

Although called soft brood selection [Tackett & Carmi, 1994], this technique uses no unusual selection methods (see 2.4.2) but instead is a modified recombination operator. Two parents are chosen at random, and a ‘brood’ of offspring is produced using the conventional crossover operator. A tournament is held within this brood, and the fittest offspring is reproduced into the next generation, even if it is less fit than either of the parents. This method encourages the production of fitter offspring than the parents, and is aimed at directly increasing the evolvability of the population.

2.4.1.4 Other genetic operators

Editing is an operation designed to reduce the complexity of program trees. It works by applying a series of domain-dependent editing rules to a genome to remove obsolete code. For instance, the code \( \text{AND}(\text{True}, \text{True}) \) can be easily matched and reduced to a \( \text{True} \) with no phenotypic effect. Similar operations can be applied to arithmetic expressions and other Boolean expressions. This operation can either be used cosmetically, to reduce the size of a program tree when being displayed for humans, or in a run to try and restrict tree growth. In a brief experiment, Koza shows that use of the edit operation has little or no effect upon the chances of GP solving a simple problem such as the 6-multiplexer problem [Koza, 1992, section 25.5]. [Soule, Foster and Dickinson, 1996] use the editing operation to try and stop program growth in GP, but it proved to be ineffective, as GP just evolved functional but non-executed code instead.

[Kinnear, 1993] uses the hoist operator in an experiment to evolve sorting networks with GP. Hoist selects a random subtree within an individual and returns a copy of that subtree as a new individual, ‘promoting’ the subtree to program status. This works best in untyped GP. Kinnear’s work shows that, while the operator is not particularly advantageous to overall performance, it does have a role to play in controlling the size of program trees.

2.4.2 Selection methods and convergence

Selection is the process by which better solutions are favoured in the reproductive phase of genetic search. A selection method uses fitness values to determine how many of each individual should be propagated into the next population. As such it is directly responsible for the tendency of the population to converge by determining the increase in frequency of each genome over time. The designer of a GP experiment wishes to balance the tendency
of the population to converge with the likelihood of exploiting a better solution should one arise. Too much convergence leads to stagnation and sub-optimal solutions, too much caution leads to the loss of many potentially useful avenues of search.

2.4.2.1 Ranking schemes vs. fitness proportionate schemes

Selection schemes can be divided into fitness-proportionate and ranking schemes. Fitness proportionate schemes use the values produced by the fitness function directly to determine the number of each individual that get reproduced. Ranking schemes use these values to produce an ordering of individuals, which is then used to control the reproduction of programs.

If fitness proportionate selection is used, normalised fitness values are calculated for each individual in the population. These values are calculated in such a way that the sum of all normalised fitness values is equal to 1. Each individual is accordingly assigned a unique interval in the range [0,1] of width equal to its normalised fitness value. A random number generator is then used to generate a value between 0 and 1, which maps directly to a single individual, with probability equal to the normalised fitness of that program. This individual is reproduced into the new population, and the process is repeated until the population is filled.

Ranking schemes use some function of the ordering of individuals to control reproduction. The most often used is tournament selection. In tournament selection, a random process is used to pick a number of individuals from the old population. Of these individuals, the one with the highest fitness value (i.e. the best rank) is reproduced into the next population. The individuals picked are not removed from the old population. This process repeats until the new population is filled. The number of individuals picked for each tournament is called the tournament size, and is an important parameter in determining the nature of the selections produced. Too large a tournament size promotes rapid convergence, too small and the nature of the selection tends towards randomness.

2.4.2.2 Propagation of an individual with different selection schemes

The rate of convergence of a population depends to a large extent on the rate at which the fittest individuals propagate in the population through selection, although this is not the only factor. The choice of selection method is therefore an important one if convergence could be a problem. The rate of convergence can be measured by looking at the propagation rate of the fittest individual in a population after a selection has taken place.
It is this individual that, assuming that the population is going to converge, will dominate the population with its genotype or phenotypically similar variants.

Propagation has been studied closely when applied to schemata (short sections of genome which contribute highly to the fitness of individuals containing them, see section 2.7.1). The aim of this section is however to give an informal notion of the rate at which convergence occurs based purely on leading edge, dominating, individuals and selection methods, to give some idea how these methods affect convergence.

If fitness proportionate selection is used, the fittest individual will reproduce in proportion to its normalised fitness value. So, for population size $M$, total normalised fitness of 1, and normalised fitness value of the best individual $N_{\text{fittest}}$, the number of individuals reproduced is $N_{\text{fittest}} \cdot M$ (rounded down to an integer). The frequency of this individual increases exponentially by a factor dependent on its normalised fitness value and those of the rest of the population. As the frequency of this dominant individual increases, its fitness advantage decreases as the average fitness of the population rises toward its peak fitness.

For tournament selection, the fittest individual will win every tournament in which it is entered. The chances of this individual being selected for any tournament are $1/M$, but with a tournament size of $T$ (that is, $T$ individuals are chosen randomly to participate in each tournament), this becomes $T/M$. Since there are $M$ tournaments held in each selection, this means a total of $T$ tournaments will be won by the fittest individual, so this individual will reproduce $T$ times.

So, for a single selection phase, the number of instances of the fittest individual (previously present only once in the population) becomes $T$. The population will become saturated with instances of this individual in $\log_T M$ generations using this selection method, assuming the effects of the genetic operators do not change the fitness of each instance in any substantial way. Even for a small tournament size such as 2 (the smallest possible value) this means that a population of size 1000 would become saturated in only 10 generations if no novelty were introduced via the genetic operators. For fitness proportionate selection, the rate of increase would depend on the actual fitness values, which would tend to homogenise over time as the fittest individuals dominate the population.

For a given problem domain, fitness function scaling can be used to affect the rate
of convergence if fitness-proportionate selection is going to be used. That is, since individuals are rewarded according to how well they do in absolute terms, changing these terms alters the advantage held by fitter individuals and affects the rate of convergence accordingly. For tournament selection and other ranking schemes, individuals are rewarded for performance in relation to the rest of the population. Here, it only matters that the individual is better, not how much better it is. For these reasons, a rank-based selection method is often popular as the designer does not have to pay attention to scaling of the values produced by the fitness function. In this thesis all experiments use tournament selection with a tournament size of 2 which gives the slowest possible rate of convergence.

An analysis of various selection methods, and their takeover times (the time taken for a population to converge once a particularly strong individual has emerged, if no genetic operators are used), can be found in [Goldberg & Deb, 1991].

[Ryan, 1994] introduces a novel selection method based on multiple criteria. A rank-based method, it uses two ordered lists of individuals which are based on separate fitness criteria. One list uses a traditional fitness function (i.e. the ability of the individual to solve the problem) whilst the other is based on a modified fitness function with a size factor incorporated. Breeding is then done between individuals in the top sections of each list, i.e. high-performing individuals are bred with good performers that are also short. This helps to maintain diversity by providing two different gene pools from which to produce new individuals. In addition to increasing performance on a sorting network problem, this helped control program tree sizes (see section 2.9.6).

2.4.3 Other methods
Methods which operate at the individual level, but do not fall into the categories of genetic operators or selection methods, include crowding and sharing. Both techniques are aimed at reducing the rate of convergence by sustaining a diverse population. Crowding [De Jong, 1975] uses overlapping generations in a similar way to the Steady State GA (see section 2.4.4.2) but instead of replacing individuals of low fitness new individuals take the place of those that are genotypically similar to the ones being reproduced. This is not done globally but rather on a portion of the population parameterised by a crowding factor, which is the number of individuals compared each time. Sharing [Goldberg & Richardson, 1987] uses genotypical similarity to penalise
fitness, so that multiple copies of the same genome are selected out by the normal methods.

2.4.4 Population-level techniques to reduce convergence

Whilst genetic operators and selection methods work on the individual programs themselves, or their fitness values, other methods adapt the underlying structure of the population-based search process to try and improve performance. The two main methods used which operate on populations in a different way are the Steady State GA (SSGA) and deme populations (otherwise known as geographic isolation). Both take their inspiration from a closer adherence to the metaphor with evolution in the natural environment.

2.4.4.1 Demes

In nature populations of a particular species do not exist in a single large body of individuals, competing globally with each other for resources. A potentially more accurate model would have a number of sub-populations of the species, existing for the most part in geographic isolation from other sub-populations, each evolving in a slightly different direction in response to differing environmental conditions.

In terms of a genetic algorithm, the sub-division of a population may be achieved in a number of ways. One way, the "island" model, is to divide the population into chunks of smaller size, say a population of 500 would become ten populations of 50 [Juille & Pollack, 1996]. Each sub-population would evolve almost independently of the others, but after selection an optional phase of migration occurs. This involves the swapping of a few individuals from one sub-population to another, to represent the migration of animals between populations through chance. The population is allowed to explore several different avenues simultaneously through the isolation provided, whilst still allowing the good genomes to spread out and grow. A second method is similar but works locally. Each individual is placed on a grid-world, with some number of neighbours. This world is often toroidal, i.e. it wraps round at the edges. During the selection phases tournaments would be held between individuals geographically close to each other. Solutions are also only allowed to mate with their neighbours, i.e. with other individuals within a specified 'distance' in the world. A phase of migration is added to allow good solutions to spread their genotypes into different parts of the population. This may produce localised groups of specialised solutions, which can develop down a particular path, but can still diffuse...
through the population if they prove to be better than their neighbours. The island model is particularly suited to large-scale parallelisation, as each sub-population can be administered by a different machine or processor with little overhead. Communication by migration takes place after each generation has been evaluated, and individuals are sent between machines.

2.4.4.2 Steady state GAs

The model of population evolution used in conventional GA search is a generational model, that is, it replaces entire populations at one time. This is in stark contrast to the natural world, where the birth and death of individual organisms is a continuous process. The fitness of an individual plays a significant part in determining only that individual’s chances of survival and is largely independent of the prowess of other individuals. The processes of death and reproduction occur at the individual level and are based largely on that individual’s chances of survival (i.e. fitness).

A Steady State GA (SSGA) [Syswerda, 1991] determines the makeup of a population at the individual level, rather than for a whole population at a time. Rather than select M individuals and then apply genetic operators over all M, the SSGA produces one or two new individuals at a time, and inserts them into the existing population. These new individuals are picked from the existing population according to selection criteria similar to those that apply in the generational model, and may be altered by one of the genetic operators. To maintain the population size, the new programs replace older individuals based on fitness criteria (i.e. poorer programs stand a larger chance of being removed).

The population in a SSGA is updated frequently and to a small degree only. Each time new individuals are inserted, they are evaluated and the fitness distribution of the entire population updated. This can mean slightly more overhead for the GP system, but the extra computation is usually insignificant compared to that required by fitness evaluations. When comparing SSGA results with traditional GA experiments, the term generation equivalent is used to refer to the number of replacements of individuals that would be equivalent to replacing a single generation in panmictic GAs.

2.4.4.3 Dynamic fitness functions

The fitness function used in a genetic search is usually a static one. It may consist of a number of fitness cases (samples of training data), over which the program is rated, but
often the cases are the same every generation. Convergence can be delayed by shifting the evolutionary goalposts on a regular basis, giving the individuals new problems to solve which differ slightly from those used previously (while retaining the essential properties that define the problem domain). Strategies which involve a random selection of fitness cases each generation, from a pool of cases, can help prevent the search finding too-specific solutions to the problem (known as overfitting). We normally want to be sure about certain properties of our training set, however, so such pools are inevitably limited in size. The search therefore is directed in a very constrained way to ensure the maintenance of such properties.

A more complex method of adapting the fitness function over the duration of the run is Limited Error Fitness (LEF, [Gathercole & Ross, 1997]), which dynamically adapts the order of the training samples used and the maximum permitted error for each genetic program, based on the current best performance in the population. This allows the adaptive scaling of the hardness of the problem, and also lets the population 'kick-start' in the event of failure if the problem becomes too hard. This technique permits a longer search process by keeping the population out of ruts in the search space, constantly pushing it in slightly different directions.

2.4.4.4 Co-evolution

Co-evolution provides a way to provide a large number of fitness cases to drive the search, and to make the cases get more difficult as the search progresses, to force the evolution of more general and more accomplished solutions.

Co-evolution is a simulation of the evolutionary arms races common in natural ecosystems. In particular, the arms race between predator and prey is a common example of two evolving species which determine each other's survival. Each species acts as a fitness function for the other: a good predator will catch lots of prey and survive, a good prey is more likely to evade predators and survive long enough to reproduce. Each new feature developed by one species forces the development of a counter measure by the other. This co-evolution can be used in evolutionary computation also. The fitness cases for a particular problem are themselves evolved, and rated upon how poorly the candidate solutions solve them. They are then bred and selected in the same way as the programs that are trying to solve them. This allows the fitness cases, once good solutions to them have evolved, to themselves evolve to exploit idiosyncrasies in the algorithms being used.

37
by the candidate solutions. One particularly good example of this is a sorting network which co-evolves with its test cases [Hillis, 1992].

Such an approach can be an elegant way to avoid stagnation in populations caused by premature convergence, by driving the evolution in different directions over time. This can also help evolution to find better, more general solutions to problems. Whilst co-evolution lets the designer avoid the need to produce possibly large bodies of fitness cases, it involves extra effort when designing the problem suite. In addition to the considerable design effort required to produce a search of the ‘forward’ problem (the desired candidate solutions), additional effort needs to be put into implementing the ‘backward’ problem (evolving fitness cases to penalise solutions to the forward problem) as well. This more than doubles the effort required in the design phase, since as well as the interactions between primitive set and fitness function, the interactions between the two populations need to be considered. It is necessary to ensure that the backward problem is constrained so that it produces legal fitness cases, and also that these cases will prove sufficiently challenging for the forward problem. Finally, many problems may simply not be amenable to implementation as a co-evolving population as there may not be a way to evolve progressively harder fitness cases. For this reason, co-evolution is not often used in problem-solving oriented domains, although it is very popular in ecosystem simulations [Cliff & Miller, 1996].

2.5 Modularisation

For GP to be able to solve bigger real-world, complex problems, it needs to be able to scale up the size of the solutions it can produce. Genetic search, which can be used on enormous search spaces, has been shown to be effective as a general purpose search mechanism. It has problems, however, sustaining an effective search for a long period of time due to obstacles such as premature convergence. For problems which demand a long period of search, where there can be an extended period of progression in the solution space as fitter individuals are discovered, genetic search can converge too quickly on a sub-optimal solution. Whilst the techniques detailed in section 2.4 can help sustain the process, other techniques have been developed to cut short the search time necessary to solve a larger problem. These techniques seek to increase the power of GP to produce more powerful solutions, which perform more operations on the input whilst still remaining parsimonious (in GP terms, parsimony refers to the desire to produce compact
trees with fewer nodes, which are more easily understood and analysed). This is achieved through abstraction in the program trees, finding code that can be re-used many times over within the same program.

Currently in GP there are three approaches to abstraction, which are detailed in the remainder of this section. They differ in the amount of effort required on the designers part, and in the scope of the abstractions produced. These approaches all concentrate on the abstraction of procedure within the program trees, however. The other kind of abstraction used by human programmers is that of data abstraction, which mirrors the conceptual abstractions used when we analyse and break down a problem. Work on using data abstraction within GP is presented in Chapter 6 of this thesis.

2.5.1 *Automatically defined functions*

Automatically Defined Functions (ADFs) [Koza, 1994] are inspired by structured programming methods where code fragments in the program are encapsulated in distinct modules. Using ADFs allows GP to exploit this encapsulation and re-usability by evolving subroutines as separate code trees, along with the main program tree. When using ADFs with GP, each program tree is supplemented with a number of separate trees each of which implements a single ADF. Each ADF tree is local to a single individual, and is not available to other individuals in the population. Calls to these ADFs are included in the primitive set of the main program tree for that individual, allowing their use in the main program tree. Each ADF is a separate function in its own right, with a primitive set of its own, that can take a number of formal parameters. These parameters are supplied by the main program tree as subtrees of a particular call to the ADF, the same way as with any other element of the function set. During the evolutionary process, the main program tree and the ADF trees can be altered by the genetic operators. The idea is that the evolution of the subroutines, and the use made of them, is performed simultaneously.

Care must be taken with ADFs to preclude the possibility of recursive calls, i.e. if we have two ADFs in a program, called ADF0 and ADF1, we cannot allow ADF0 to call ADF1 whilst simultaneously allowing ADF1 to call ADF0. This could result in an infinite recursion. An often-used solution to this problem is to impose a hierarchy on the ADF trees, allowing some to call others but constraining the primitive sets so that the possibility of recursion never arises. For example, with 3 ADF trees per program, we could allow ADF0 to call ADF1 and ADF2, and allow ADF1 to call ADF2, but not allow
ADF2 to call any of the other ADF functions. No ADF is usually allowed to call itself. The main program tree would be allowed to call any ADF tree. This would prevent any possibility of recursion whilst still allowing for a limited hierarchy of abstraction to form.

Restrictions on the genetic operators are necessary for ADFs to remain well-defined in all cases. The main restriction is that if crossover is to occur between two programs, the same trees in each program must be used. That is, the main program tree of one parent can only be crossed with the main program tree of the second parent, or a particular ADF tree can only be crossed with the tree of the corresponding ADF in the other parent. This constraint is called branch typing, and is used to ensure correctness of the swapped trees. ADF trees often have references to formal parameters in them, and these are only guaranteed to be meaningful if the crossover occurs between trees with the same prototypes.

Koza shows in [Koza, 1994] that ADFs can be useful over a wide range of problem domains. They can help the evolution proceed more quickly and allow GP to solve problems that it could not solve if no ADFs were present. The design effort which goes into producing the primitive set for a particular problem without ADFs has to be repeated when using ADFs, to decide what role (if a role is to be specified) each ADF should take. This means possibly designing several primitive sets for a given problem. In later experiments, Koza shows that GP can be used to evolve both the number and structure of ADF trees in a program [Koza, 1994, Chapter 21].

2.5.2 Module acquisition
Module Acquisition (MA) [Angeline, 1993] is an abstraction mechanism that requires no design effort on the part of the designer. It extends the work done by the ordinary GP mechanism to include automatic abstraction of blocks of code.

Two new genetic operators are added to conventional GP, called \textit{compress} and \textit{expand}. The compression operator takes a subtree from a program, and stores it in a global library of routines. The subtree is replaced with a single node in the program tree. This node may be a terminal or may take some parameters which are substituted for the terminals in the subtree that is removed. The module thus created is stored in a library and is available for use by any program in the population.

The expand operator does the opposite. A library module is removed from the library and all references to it are replaced with copies of the module’s code. This allows
the library to remain of a fixed size and permits the code to be modified by the normal search process.

Whilst compressed as library routines, the code in the module is immune to possibly destructive effects caused by crossover as it is now atomic. This can have advantages if the module’s code is ‘good’ i.e. it contributes to a high fitness score in the individuals that contain it. Although use of this technique is not now thought to give an advantage in terms of finding a more fit solution [Kinnear, 1994], it does help by producing more structured solutions than those produced without it.

2.5.3 Adaptive representation
The operators used in Module Acquisition are, like the traditional operators of mutation and crossover, driven by random factors. The action of the selection process is still used to provide more copies of fitter individuals, and the compress and expand operators rely on this to modularise ‘fit’ subtrees of code (ones that occur repeatedly in high-fitness individuals). [Rosca, 1996] uses a modularising operator to extract blocks of code and add them to the primitive set during a run. This is done on an epochal basis, every few generations, and examines all new blocks that have arisen since the last epoch. The blocks are tested for fitness using a cut-down version of the same fitness function used to test the program trees, and the best ones are incorporated into the primitive set. This method is based on the assumption that trees which can solve simple versions of the problem will be useful to other trees which are trying to solve bigger instances of that problem. This was demonstrated when GP with AR solved high-order instances of the even and odd parity problems, that GP alone could not solve.

2.6 Expressive power of GP
One of the most important factors to consider when applying GP to a problem is whether the primitive set has the expressive power to solve the problem. The ability of GP to find a solution depends on the ability of the primitive set to express the computation needed to take the inputs and produce the desired output. It is notionally very simple to produce a primitive set that can perform the required computation, all that is needed is one that is Turing complete and, assuming that the problem can be solved by a Turing machine, we have the expressive power needed.

This is not necessarily a good approach to increasing the expressive power,
however. Using a Turing complete primitive set alone ignores the benefits to be gained from decades of human endeavour into language design, software engineering principles, computational efficiency or algorithmic research. Moreover, it could be so hopelessly low-level that no solution could ever be expected to materialise (see section 2.7.2).

We can adopt lessons from human approaches to programming in the genetic programming paradigm. The following sections detail several approaches used to augment the ordinary GP process to increase the power of the programs that are produced. In addition, techniques designed to increase modularisation such as those detailed in the previous section, can often help to increase the ability of a primitive set to solve a problem by extending the ‘reach’ of the search i.e. the maximum distance in the solution space that can be travelled within the lifetime of a single run. Several examples of ADFs increasing the order of problem that can be solved, and hence the expressive power of the primitive set, can be found in [Koza, 1992] and [Koza, 1994].

2.6.1 Memory

One way to increase the power of programs produced with GP is to give them a mechanism for storing results produced by one section of the program, and then referencing them later on. Usually GP is entirely functional, i.e. there are no side-effects when evaluating a tree, so this facility does not exist except in the normal semantics of return values being placed on the program stack. With appropriate primitives and external storage, GP programs can evolve to read and write memory cells. This means programs can have state.

[Teller, 1994] adds indexed memory to GP with two primitives, Read and Write. Read takes an index \( Y \) and returns the contents of a global array \( \text{Memory}[Y] \). Write takes two arguments \( X, Y \) and puts \( X \) into \( \text{Memory}[Y] \), returning the old value of \( \text{Memory}[Y] \). Each argument is a terminal taken from a range guaranteed to be within the size of the array, which was fixed at 20 cells. The use of state allowed results in a problem domain involving the evolution of agents to move boxes within a grid world, a problem that was designed to be insoluble without some kind of internal state being available to the programs.

The use of unstructured memory allows agents to store state and refer back to it later. However, it is hard to use due to the large number of points at which state can be stored and retrieved, i.e. the large number of memory cells. A more structured approach is
taken by Langdon, who evolves structured memory elements such as queues, stacks and lists from an indexed memory array and some access primitives [Langdon, 1995]. A further set of experiments shows that such structures can be used by GP to solve a range of problems that require state [Langdon, 1996], and that GP with indexed memory could not solve these example problems.

2.6.2 Turing completeness, recursion and iteration

The range of computations that can be performed by a genetic program is constrained by the primitive set from which it is formed. To increase the power of the program, it is necessary to increase the power of the primitive set. [Teller, 1994a] shows that, with the addition of iteration and a mechanism to read and write indexed memory, a GP system can produce any algorithm and is thus Turing complete. This means that no computation is theoretically outside the bounds of GP’s capabilities. In practice this is often not the case, as any Turing machine evolved with GP is bound by the same constraints that ordinary Turing machines are, i.e. we would not know whether our random machine would ever halt and when instantiated physically we would need possibly infinite memory resources. This work does however demonstrate how flexible the paradigm of GP could be.

[Koza, 1992, Chapter 18] uses an iteration primitive to solve a block-stacking problem and to evolve a summation operator for a sequence of numbers. Koza also uses a very highly constrained recursion operator to evolve a program that generates the Fibonacci sequence, only allowing the recursion to take place if it calls a lower-order instance of the function (i.e. with an argument closer to 0). [Brave, 1996] allows ADFs to call themselves on a tree-search problem, using the depth of the input trees as a base case for the recursion (ADFs are not allowed to refer to nodes beyond the maximum depth of the tree being used as input data). Use of recursive ADFs was reported to outperform ordinary GP and ordinary GP with ADFs on this problem. [Wong & Leung, 1996] take a different approach when evolving recursive solutions to the even-n parity problem, and restrict the total number of nodes that can be executed by a program tree to guarantee termination.

2.6.3 Strongly typed GP

To increase the power of the solutions evolved by GP, we should take advantage of the practices and techniques developed by human programmers over the years to increase the
power of human-written programs. One such development is the idea of creating and manipulating entities in programs that reflect the nature of the entities present in the real-world problem which the program is trying to model. By capturing such aspects of the problem in different *types* of data, we can make the program both easier to understand and easier to create, since the structures we manipulate in code map directly to our knowledge of the problem domain.

Montana [Montana, 1995] introduces the concept of data types to GP and calls the resulting system Strongly Typed Genetic Programming (STGP). In STGP, each element in the primitive set is associated with a given data type, just as it might be in a hand-coded program. This type is the type which that node returns to its parent or, in the case of the root node, the type returned as the candidate solution to the problem. In addition, each function has types associated with each element of its parameter list, constraining the types which subtrees of this primitive can take on.

Crossover and mutation are restricted so that they operate only upon subtrees headed by nodes of identical type. In Montana’s implementation, crossover starts by picking a random subtree from the first parent. The crossover point for the second parent is then chosen randomly from all nodes in the second parent that have the correct type. For the mutation operator, it is necessary to generate a replacement subtree that has the same return type as that of the subtree to be replaced. These constraints ensure type safety of the program tree after each genetic operation.

Montana goes further than a simple type system and introduces the notion of generic types and generic functions. Generic types are equivalence classes in the type system. Any member of one equivalence class may stand in for any other member of that class and the program will still be type safe. Montana uses a mathematical system of vectors and matrices as his example - it would be pointless to define a distinct type for each possible size of vector when one type could handle the job adequately. So, instead of defining unique \( \text{VEC-2}, \text{VEC-3} \ldots \text{VEC-n} \) classes, a generic class \( \text{VEC-k} \) is defined with a parameterised size argument. Thus, a vector of any size is still an instance of class \( \text{VEC-k} \).

Generic functions are defined to cope with the variations possible within a generic class, and functions may also be defined to take parameters of specific instances of generic classes, i.e. specific sizes of \( \text{VEC-k} \).

The use of arbitrary data types, and the potential for generic type systems, allows
a much greater range of expression within a genetic program. There are many problem
domains, particularly more complex ones, that are extremely hard to represent
convincingly with a mono-typed system (the template matching problem in Chapter 5 is an
example of such a domain). Strongly typed program trees allow STGP to capture a much
more complex situation in a smaller program tree, and to make manipulations of data in
ways that fit well with the dynamics of the particular problem. A significant factor is the
use of container classes in the type system. These could be generic types, and allow the
tree to generate arbitrary amounts of information which can then be stored and processed
by other tree fragments.

The practical benefits of STGP above GP were shown in [Haynes et al., 1995].
The authors applied STGP to a Distributed AI problem where multiple homogenous
predator agents co-operate to capture a prey agent on a grid world. STGP was able to
evolve programs to control the predators that performed competitively with the manually
derived algorithms. It was reported that STGP produced better performing programs that
were easier to understand than those produced by mono-typed GP on the same problem.

A further enhancement to STGP is the introduction of type inheritance hierarchies
[Haynes, Schoenefeld & Wainright, 1996]. The notion of inherited type comes from
Object Oriented Programming, and complements the notion of generic typing. With
generic typing, an equivalence class is a set of types where any type in the class can be
substituted for any other. Inheritance hierarchies are directed acyclic graphs of types. A
particular type can only be substituted for another in the type system if there exists a path
in the graph from the node for the required type to that of the inherited type. Nodes may
inherit from any number of other types simultaneously (multiple inheritance), and may be
inherited by any number of other types. This is an implementation of inheritance from
OOP, but this description places no requirements that the inherited types actually inherit
(in the OOP sense) the behaviour of the superclasses. It merely specifies the possible types
that may be inserted at particular points in the program tree. A convenient implementation
is to match this system to the underlying inheritance systems of the execution language.
The authors demonstrate the use of inheritance and STGP on a clique detection problem,
for single inheritance only, and found that while plain STGP failed on their problem,
STGP with inheritance produced a solution.

STGP works by cutting down the search space that needs to be explored in a
number of ways. Firstly, the number of options available to the tree generation process at
any given point are limited by the number of primitives available of the required type. In un-typed GP the closure property requires that any function is well-defined for all possible values that could be returned by its parameters. The closure property for STGP is modified to be restricted to a particular data type, i.e. any function that needs to take a real valued number as an argument should be able to accept any value that could be produced by elements of the primitive set that return a real number. This immediately cuts down the number of branches in the search space by constraining the tree construction process.

Secondly, each primitive can do a lot more work in the space of a single node, work that might need a large subtree to implement in an un-typed system. This is due to the richness of the data types that can be passed around, and the comparative paucity of a single number as a way of representing the results of all the computation that occurred in a program tree. This extra expressive power means that trees can often be much smaller in STGP than equivalent trees in GP.

A lot of work in this thesis uses STGP. The use of this approach allows a natural way of producing solutions to problems that are of varying size. Taking the domain of signal processing as an example, applying an un-typed program to a training input signal would require that the problem be set up so that the candidate program was applied once to every element of the signal. Some interpretation of the outputs would then be necessary to get a fitness value out. In other words, the GP doesn’t evolve a signal processing algorithm but some kind of signal transformation, and it is up to the designer to analyse the outputs. This is the set-up used in the edge detection experiments detailed in Chapter 4. It would be possible, using STGP, to use a primitive set that, with a single execution, operates directly on the whole signal regardless of size to produce some result that is directly applicable to the analysis required by the problem. Such flexibility, and the ability to match the program domain so closely to the problem domain, makes STGP a powerful extension to genetic programming.

The techniques of ADFs and strong typing can be combined quite easily. No extra constraints are needed above and beyond those already imposed by ordinary STGP and GP with ADFs. Crossover is already constrained to only swap between identical ADF branches of each parent program, to ensure safety with formal parameter names. This branch typing [Koza, 1992] ensures naming safety in any subtrees which are swapped and might contain references to formal parameters. Since each individual in a population has
identically defined prototypes for both the main, value returning, branch and all ADF branches, this branch typing also ensures type safety under crossover. Mutation needs no modification above what is necessary for STGP, so ADFs can be used without any extra modifications or constraints above those required by the two techniques separately. To date, no applications of STGP with ADFs have been seen, although an implementation was produced for this thesis. The advantage of combining the two approaches has yet to be demonstrated, therefore.

2.6.3.1 Evolving data structures

One of the most interesting side effects of employing STGP is that the designer is no longer restricted to returning just a single number from the candidate program (see Chapters 5 and 6 for examples). While this may seem an obvious and seemingly trivial consequence of the move to a full type system, it has wider implications.

GP is often seen as generating programs that solve problems. The emphasis here is on passing in some inputs and getting an output that hopefully is the result of some useful analysis on the part of the program. Often, with an un-typed system, this is all the information that could be hoped for. Such results would then have to be interpreted by the fitness function. For instance, a boolean decision problem would return a number that would be interpreted as True if it was greater than zero, and False otherwise. The emphasis is then on producing a program that returns some function of the training data.

With STGP this approach is not the only one that can be taken. The use of container classes within the primitive set allows programs that return complex and dynamic data structures instead of single numbers. This allows GP to be applied to, for example, problems where the required task could be to capture some property of a set of data, and return the abstracted form. Such problems could not be addressed by un-typed GP because the representation is not sufficiently rich. This opportunity is a change in emphasis on the role of GP in problem solving, from producing some operation on a piece of training data to producing data itself. Examples of this approach can be found in Chapters 5 and 6.

The use of complex data structures as return values has implications for the designer. When using un-typed GP very little information is returned. This means that the fitness functions used are necessarily very simple, as very little analysis can be done on a single number without an implied semantics. An example here would be problems such as
classification, where un-typed GP not only has to discriminate between classes of data based on training examples, but also has to learn the semantics implicit in the fitness function stating which ranges of numerical return values should map to which classifications. By encoding the classifications as terminals in the primitive set, using a strongly typed system, the semantics is made explicit and the GP only needs to map the training data to the correct classification.

To get a more complicated result from a single genetic program, the program is often run many times over a range of fitness cases. The fitness function for such situations often measures 'hits', which are correct answers for a given input case. Programs are rated by the number of hits they achieve. In other cases, the program is run once for each element of compound input data. For instance, a signal processing problem domain as in Chapter 7 applies the program to every element of the input signal, to produce an output signal by joining the individual responses together. This output is then analysed to produce some fitness measure. When the data being output (either through a single complex data type or the amalgamation of many simple readings) is so complex, the function to analyse this output and produce a fitness value can also be more complex than a function for simple outputs. This makes the task of designing a function that will drive the search in the correct way, far more difficult. Genetic search tends to exploit idiosyncrasies in the fitness function to gain greater fitness values, and a more complex function will have many more 'holes' and gaps through which the search will pass. This means that it can be extremely hard to debug the search process and get it moving in the right direction. This problem is exacerbated by the relatively long run-times that tend to be associated with more complex problems, increasing the length of time it takes to get a useful result.

2.6.4 Flexible program representations

The representation of programs as trees often leaves a lot to be desired in computational terms. Although constructs such as looping can be represented as trees, they are not executed as trees (since control flow does not proceed in the usual depth-first manner for looping), and rely on some interpretation to be run correctly. The ability to generate recursive and iterative code directly could be useful to increase the power of the programs produced. Furthermore, it can be useful to evolve the structure of the program in terms of control flow, along with the contents of each node, when using GP to evolve
computational structures such as recurrent neural nets, for example, where the program directly encodes the structure of the network [Esparcia-Alcazar & Sharman, 1997]. Although tree-based representations are still dominant, other representations do exist.

In the PADO system [Teller, 1995] programs consist of an arbitrarily directed graph rather than a tree. Each node contains an action, which acts upon a global stack, taking arguments as necessary and placing a value back on the stack, and a branch-decision, which decides which of the outgoing arcs from the node to follow. Both the action and the branch-decision are evolved functions which have access to the program's indexed memory and previous state. This gives the programs extremely complex dynamics, as they are able to execute continuously and change the course of their executions based on previous operations. To cope with the problems caused by programs running interminably, the programs are killed after a specified amount of time and the contents of a number of memory cells taken as the program's result.

In Parallel Distributed GP (PDGP) programs are represented as directed graphs on a 'grid' [Poli, 1997]. Each node represents a function or terminal from the primitive set, and edges represent the flow of control or results between nodes. Multiple edges from a node mean that the sub-graph headed by that node is called multiple times within the program. Each sub-graph need only be executed once, and it then caches its result, so execution of the program can be more efficient than with trees. The representation encourages re-use of subtrees within the same program and is therefore very efficient and compact. Such a representation can be made extremely flexible when labelling schemes are used on edges, i.e. adding weights to edges can allow the direct evolution of a neural network. This representation was shown to be extremely effective where some kind of structural regularity is required in the solutions, often being hundreds of times more efficient than ordinary GP in terms of CPU time and program size.

Stack-based GP [Perkis, 1994] encodes genomes as linear structures which execute in a stack-based virtual machine. Primitives are executed in sequence and push and pop results and arguments from the stack as needed. This is more like the imperative model of programming, compared to Koza's functional model. Performance was shown to equal or surpass tree-based GP in three simple problems, usually requiring less effort to solve the problem when stack-based methods were used.
2.7 Role of the primitive set

The successful application of GP to a problem domain hinges on the interaction of two problem-specific elements, the primitive set and the fitness function. If we take the simplistic view of search within a multidimensional space such as is often used to explain GA search, we can look at the primitive set (in the case of a simple GA, the binary alphabet or a number of real values) as constituting all but one of the dimensions of the space. The fitness function maps indices for these dimensions to real numbers, giving the final dimension of the search space. The nature of the search space depends on both primitive set and fitness function, so both need to be carefully designed. A change of primitive set effectively turns the problem into a new one, since it changes the way in which novel solutions to the problem are suggested to the fitness function. The fitness function is equally important, in so far as a change in this function is a change in the mapping from genome to fitness, which could turn a previously good solution into a bad one or vice versa.

2.7.1 Search spaces in GP

The convenient way of thinking about GA search, as a search through a multidimensional 'landscape', does not translate well to GP. The dimensionality of the search space in GP is not fixed, since trees can be of different size. The size of each dimension is decided by the size of the primitive set, and this may vary in cases where some modularisation method is used. Practical implementations of GP tend to put an upper bound on the size of a program tree, so this can be used as an upper bound on the dimensionality of the search space, but it gives us no useful metaphor with which to visualise the search. GP search relies on the theory of GA search, and it is not clear that the theory underlying GA search applies to GP.

Holland derived the Schema Theorem for GAs [Holland, 1992] which shows that GA search implicitly samples many solution strategies in parallel whilst searching, and promotes good strategies through propagation of individuals containing those strategies. The strategies, called schemata, are templates representing entire groups of genes by specifying only some of the values that the genome should have. The schema theorem is generally accepted as a good explanation for the power of GA search. Several attempts have been made to adapt this theorem to GP, but with only partial success. Although several versions of the theorem have been produced, each attempt either makes some
change to the standard GP method to preserve a formalism, or was less exact on the
degree of propagation of high-quality schema. A good summary of work on GP schema
theory can be found in [Poli & Langdon, 1997a].

The schema theorem illustrates that much of the power of genetic search comes
from a correlation between parental fitness and the fitness of offspring produced. This
makes common sense - a fit parent is going to be selected preferentially so further
progress would be expected to come from its multiple descendants in the new population.
If the search process is not to stagnate, or worse to actually produce less fit individuals,
the chance of producing offspring at least as fit as the parent needs to be as high as
possible. This is precisely the property of evolvability referred to in [Wagner & Altenberg,
1994].

2.7.2 Desirable properties of a search space
If we keep to the image of a search space as used to visualise GA search, good
evolvability in GP is indicated by a smooth landscape. This is linked to the concept of
progression, which requires that a small change in genotype produces a small change in
phenotype, so that a steady progress from early solution, through intermediate quality
solutions, to an eventual high-quality solution can be made. In other words, altering an
individual in a small way, say by use of a genetic operator, shouldn’t so drastically alter
the behaviour of the individual that it produces a wildly different fitness value. Generally,
genetic operations are destructive, and are detrimental to fitness by moving the individual
some distance away from a ‘good’ place in the search space. Progression would be better
if the genetic operators only changed the genomes by a small amount, moving them a
smaller distance in the search space, and if that search space were fairly smooth. This of
course is traded off against the chance of not escaping from a strong attractor into other,
possibly more lucrative, areas of solution. One way in which the effect of an operator
could be limited is by increasing the likelihood that operators work on terminals (and
therefore very small subtrees consisting of one node in the genome), minimising the
amount of genotypic difference between the parent and child. Unfortunately, in work
unrelated to the notion of progression, [Angeline, 1996] suggests that the ideal frequency
of ‘leaf selection’ (the chance that a terminal will be selected as the subtree to be operated
upon by a genetic operator) used in GP varies with problem domain, which is to be
expected if the primitive sets change so much from problem to problem that we don’t
maintain a constant genotype to phenotype mapping.

A large number of genetic operations applied to a group of identical genotypes will tend to spread those individuals out around the place in the search space occupied by the original genome. If the search space has high-frequency components then it is more likely that the many of the new variants will be vastly worse off than before and the operations will be wasted. Exploitation of a potentially lucrative area of the space is thus made less effective. A better search space would be dominated by low-frequency components, so that any movement within the space would result in a smaller change in fitness and thus increase the chances of the new variants surviving to explore other parts of the space.

High frequency components in the fitness function produce many optima in the space - it is said to have a high modality. Landscapes of high modality are hard to search because it is more likely that the search could get stuck on a local optimum that is not the global optimum and thus converge to a sub-optimal solution. Also, in high-modal landscapes, good fitness values tend to occur in very small locales within the space, which can easily be overlooked by the search. Additionally, solutions near such a high-frequency peak are more likely to produce less fit offspring since even a small displacement in the solution space could miss the peak entirely. The ideal landscape would be a uni-modal landscape with smooth sides so that the search could easily find the maximum once it was within the basin of attraction of the peak. These sorts of landscapes are so easy to search, however, that a simple hill-climbing method would probably be more appropriate [O'Reilly, 1995, Chapter 6].

The chances of finding the best possible solution with any search are to some extent determined by the number of nodes within the search space that can be explored. With genetic search, this can obviously be increased by devoting more resources to the search, either by increasing population size or by running the search for longer, or more inventive methods to ensure a better coverage of the search space (to avoid re-evaluating the same genome). This approach however can only have minimal effect in the context of problem domains that usually have huge search spaces which is the case with GP. Cutting down the size of the search space can be more beneficial as it permits a larger proportion of the space to be searched with the same effort. Techniques such as strong typing, and introducing more powerful and relevant primitives, can be of use to limit the size of the search space.
2.7.3 Sufficiency of a set of primitives

The designer needs to be sure that the primitive set used is capable of producing a solution to the problem at all, otherwise the entire enterprise may be wasted effort. This property, known as sufficiency, does not in itself guarantee that a solution will be found since the outcome of the search is non-deterministic, but it can provide a base line for designing a primitive set with the expressive power to produce a solution. This sufficiency property is a necessary precondition for a successful solution, but is not in itself sufficient to guarantee such a solution.

One way to determine the expressive power of a set of primitives is to measure the amount of effort that goes into the design and implementation of those primitives. The complexity of the primitive set need not relate to the chances of success, however: a Fourier transform may be complex but would be unlikely to increase the performance of GP in reverse-engineering a boolean function.

Another way might be to make statements about the variety of programs that the primitive set can produce. The range of, or set size of, outputs that could be produced is a useful way to see whether the primitives could generate the necessary results. Unfortunately the simplest of primitive sets is often capable of producing an almost infinite range of results so this is impracticable.

An experimental method to determine expressive power is to find the smallest possible program tree using the given primitive set, that is capable of producing an answer to the problem. The smaller the tree, the more powerful the primitives. This is a good measure but hard to obtain in practice. Many problems are optimisation problems, so the best solution may not be known beforehand and it is therefore hard to know when a good enough solution has been reached (although some arbitrary performance threshold could be used as a milestone). A bigger problem is that it can be impossible to generate all the program trees that provide the answer to the problem, so the smallest tree might not be found.

Designers rely on intuition and inspection to see if their primitives are sufficient for the target problem domain. In practice, if one is in doubt, it is easy to add more primitives to the set to increase the chances of sufficiency. This approach can lead to large primitive sets, however, which reduce the chances of the search process finding anything useful by vastly increasing the size of the space to be searched. Although the sufficiency property is necessary for a successful solution, it is rarely sufficient in itself since the
The search process used is not infinitely capable. More effort needs to be expended in ensuring a certain appropriateness of the primitive set, more simply expressed as a property of suitability. A set is suitable if it contains primitives that the designer thinks will be helpful in producing a good solution to the problem. More suitable primitives usually embody large amounts of domain-specific knowledge and make the search easier.

The previous sections in this chapter detail several approaches used to augment the ordinary GP process to ensure either sufficiency, or suitability. Although these are significant advances in GP technology, they are not for the most part employed by the majority of GP practitioners as they each require a different design methodology to canonical GP.

2.7.4 Basis functions

Ideally, the primitive set chosen to provide solutions to a problem would be able to express any possible solution available within the problem domain. The idea is that for each possible return value for a given input, there exists some combination of primitives that can produce that value. This notion is that a set of basis functions for the problem can produce for any point in the output space some point in the input space using primitives from a small set.

The concept of completeness in terms of search space coverage automatically confers the sufficiency property of a primitive set, and is therefore desirable. Whereas the sufficiency property requires that a primitive set be able to express a solution to the problem, the completeness property requires that the primitive set be able to express all possible solutions to the problem. This is often hard to achieve in all but the simplest problems. One common example of a primitive set that constitutes a basis for the search space is the boolean primitives. These are used extensively in GP literature to solve a variety of benchmark Boolean multiplexer or parity problems because these problems can be addressed with a small set of primitives that are known to be complete.

2.7.5 Effects of primitive sets upon search efficiency

The probability of finding a good solution to a problem in a single run of GP depends on the size of the search space and its amenability to exploration. The size of the search space depends directly on the size of the program trees and the size of the primitive set. Bigger trees means more possible permutations of program structure, and a larger primitive set means a larger number of variations are possible at each node in the tree. In
addition, the ratio of functions to terminals in the primitive set may impact the likely size of any trees generated either as part of an initial population or through mutation, although generation methods (see section 2.3) often exert explicit control over this characteristic. Finally, since the limiting factor on tree size is often a depth constraint, the arity of the functions available in the primitive set will also affect final tree sizes.

The chances of finding a good solution therefore improve with small primitive sets, and smaller tree sizes, since there is a smaller space to explore. Assuming that a small primitive set combined with a size restriction could still find the good solutions, this might seem to be a good heuristic to use when choosing parameters for a GP run. In practice this is overshadowed by a number of factors. The size of a GP search space is usually astronomical, and even pruning the space in terms of branching factor (the number of primitives) or restricting the maximum size of the trees still leaves a huge space to search. A trivial example, using a primitive set containing three functions (all of arity 2) and two terminals, with a maximum tree depth of 10 (much smaller than used in practice), would have $3^{2^{x-1}} \times 2^{2^y}$ potential unique trees if the full method was used for tree construction. Additionally, the use of small primitive sets can require larger program trees to produce a solution if the primitives are very simple.

The size of the primitive set influences the chances of finding a good solution in a far more direct way. By offering a large choice of primitives for the construction process to choose from when creating a node, no effort is put into constructing useful trees in terms of the semantics of the program. Additionally, the genetic operators have a smaller chance of selecting the ‘right’ subtree (i.e. one that if swapped would lead to an increase in fitness of the offspring) if all primitives were equally likely to be selected. Many genetic operations, the most valuable commodity in the resources available to the search process, are wasted in this way. A common manifestation of this is the appearance of large subtrees devoted to calculating a single value from a set of constants (random or pre-ordained) and the arithmetic operators. Powerful, problem-specific primitives that would be useful in constructing a solution may be overlooked very often because of ‘swamping’ of the primitive set by other primitives thrown in ‘for good measure’. The ideal size of primitive set to use is highly dependent on many factors, such as problem representation, search energy available (the total amount of sampling of the search space) and what enhanced techniques, if any, are to be employed.
2.7.6 Type systems in constraining search

It is in the interests of the designer to reduce the opportunity for fruitless search in a GP run. This implies that careful design of the primitive set is important and that the way in which GP is expected to construct solutions needs to be borne in mind when specifying the functions and terminals to use. In all but the most trivial problems, however, it is a very difficult task to determine \textit{a priori} the usefulness of a given primitive in the construction of solutions. Leaving useless primitives in the set means that the initial population for a given run is disadvantaged before any search has been done, and it is then up to the selection methods to remove instances of these poor schemata from the population via reduced fitness. This takes away valuable time which could otherwise be spent more profitably.

If determining the usefulness of each primitive (even disregarding epistatic interactions - changes in behaviour of one primitive due to the presence, or location, of another primitive - between primitives that complicate the procedure even further) is not feasible or reliable, then some other method of improving the starting conditions of a run is required. Placing constraints on the way the trees are constructed can help to cut down the options available to the tree generation process and the genetic operators, forcing the search along more focussed paths within the space. The constraint method may be a custom one, added to make the program trees reflect the structure of the underlying problem more accurately, or it may be a generic constraint method. Strong typing can be used as such a generic method.

With STGP, program construction and genetic operations are constrained by the type system. A subtree must be headed by a node that returns the type required by its parent, and genetic operators manipulate subtrees of identical type. In each case, the choice available in terms of primitives is much reduced, being limited to those that have the correct type. This cuts down the amount of variation considerably, and thus cuts down the size of the search space. The use of STGP to impose a structure on solutions is investigated further in Chapter 6.

2.8 Applying GP to signal processing problems

There has only been a small amount of work applying GP to image processing or signal analysis problems. This in part reflects the relative youth of GP as a separate discipline compared to, say, genetic algorithms or neural networks. GP seems to be well placed to
perform tasks and analyses that these other techniques could not in a field in which applications and problems are extremely varied.

[Johnson, Maes & Darrell, 1994] use Genetic Programming to evolve detectors for hands in silhouetted images of human figures. Using strong typing for their primitive set, they assume a fair amount of pre-processing for each fitness case, including both localisation of the silhouette and a number of statistics about it. A large amount of effort is devoted to discussion of how various fitness functions affected performance. They also manage to make some statements about the size of the search space, since they restrict the maximum depth of their trees.

[Andre, 1994] uses GP to evolve functions that would traverse an image, calling upon co-evolved detectors in the form of hit-miss matrices to guide the search. These hit-miss matrices were evolved with a 2-dimensional GA. Andre used this method to evolve functions that could discriminate between 2 letters, or recognise single digits, even in the presence of noise in the images.

[Koza, 1993] uses ADFs to show how hierarchically structured solutions can be applied to the San Mateo trail problem. This problem involves programming an artificial ant to follow a trail of food on a grid world. The trail may have gaps of one or two squares where no food is present. The ant is allowed basic local sensor information, and can either move forward or turn 90° in response to some condition. ADFs used in this context are interpreted by Koza as ‘detectors’ for certain configurations of terrain, returning positive values if the terrain matches some configuration represented by that ADF. Koza shows that using ADFs reduces the average structural complexity of a solution and the number of individuals that have to be processed by the search before a solution is found.

[Poli, 1996] applies GP to the problem of segmentation of magnetic resonance images. Using a primitive set that takes information directly from the input image, filters are evolved using primitives which are capable of emulating any image processing filter. The output of these filters is passed through a threshold decision process to binary-segment the image. The driving force of the evolution is determined by a single parameter to a function that, depending on the value of the parameter, can optimise the number of false positives found, or the number of false negatives. Performance is found to be better than a neural net approach on identical data.

[Teller & Veloso, 1995] uses the PADO language to perform face recognition
tasks on a database of face images. Although encouraging results are obtained, the authors stress that the face database used, like many in face recognition tasks, is insufficiently general to be of practical use. In particular, the face itself is rarely needed to recognise the image! [Teller & Veloso, 1995a] uses PADO to perform a discrimination task between 5 classes of image. The classes of image, consisting of different kinds of shape placed at various sizes and positions on black backgrounds, are chosen to prove the discriminating ability of PADO, and are such that the distinctions between them are conjunctions of abstract features. Additionally, discrimination can be performed on these images with noise and obstructions added. Without noise, PADO achieved respectable performances of up to 60% correct discrimination between 10 classes.

[Tackett, 1993] uses GP to assign detected image features to a ‘target’ or ‘non-target’ category. When classifying feature vectors, GP was found to outperform both a neural net classifier and a binary tree classifier on identical data, producing lower rates of false positives for identical detection rates. The best evolved program was also found to require less computation at run-time than the neural net, and still achieve better performance. A second experiment which works on primitive intensity measurements from the image, produced mixed results, outperforming the neural net only when a high probability of detection was required.

[Daida et al., 1996] discusses implementation and software assistance issues for the collaborative design of image classification algorithms. The experiments carried out as illustration are detailed further in [Daida et al., 1996a], which uses GP as an algorithm development tool, designed to assist a user. A case study is the extraction of sea-ice pressure-ridge features from synthetic aperture radar (SAR) imagery. A dynamic fitness function, where the difficulty of the training sets gradually increases over time, was used to help scalability.

[Nordin & Banzhaf, 1996] apply a linear-structured form of GP, based around primitive sets consisting of machine-code instructions, to the programmatic compression of digital signals, both sampled sound signals and 2-D raster images. The GP is a simple application of symbolic regression to the input signal, using a squared-distance error function to drive the evolution. Use of a compiling genetic programming system [Nordin, 1994] allows huge numbers of evaluations. The quality of reconstructed signals depended heavily on the time taken to evolve the model, but decompression is very quick.

[Robinson & McIlroy, 1995] apply GP to the problem of eye location in grey-level
face images. The input data from the images is restricted to a 3000 pixel block around the location of the eyes in the face image. This produced promising results over very small training sets, up to 100% true positive detection and no false positives, on a three-image training set. Over larger sets the GP performed less well, however, and could not match the performance of Neural Network techniques on similar problems.

[Koza, 1994, Chapter 15] uses a ‘turtle’ to walk over a bitmap landscape. This bitmap is to be classified either as a letter ‘L’, a letter ‘I’ or neither of the two letters. The turtle has access to the values of the pixels in the bitmap by moving over them and calling a detector primitive. The turtle uses a decision tree process, in conjunction with navigation primitives, to walk over the bitmap and decide which category a particular landscape falls into. Limitations on resources prevented the discovery of a solution without using ADFs, although it was thought that a solution was possible. Using ADFs as local detectors, and a constrained syntactic structure, perfect-scoring classifying programs were found. Further experiments showed that detectors could be made for different sizes and positions of letters, although each detector was specialised to a given combination of these factors.

[Koza, 1994, Chapter 18] interprets protein amino-acid sub-sequences as one-dimensional signals. These signals are analysed with GP to predict which sequences are *transmembrane* sequences, which are embedded in cell membranes in such a way that the protein traverses the membrane and exists in part inside, outside, and within the membrane. Supplied ratings of the hydrophobicity of each type of amino-acid are used to produce a fitness function. GP is then required to evolve an expression that returns a positive value for generally hydrophobic (transmembrane) sequences, and non-positive for non-transmembrane sequences.

[Isaka, 1997] uses GP to locate mouth corners in a small (50 x 40) image taken from pictures of faces. Processing each pixel independently using an approach based on relative intensities of surrounding pixels, GP was shown to perform comparably to a template matching approach on the same data, but with reduced computational requirements.

[Sherrah, Bogner & Bouzerdoum, 1997] use GP to pre-process measurement data prior to passing this data to a standard classification algorithm, for a number of problem domains. GP is used to reduce the dimensionality of the data and thus improve the performance of the classification algorithms over that achieved on the raw measurement data.
data.

[Winkeler & Manjunath, 1997] produce genetic programs to locate faces in images. Face samples are cut out and scaled, then pre-processed. The statistics gleaned from these segments are used as terminals in the GP which evolves an expression returning how likely a pixel is to be part of a face image. Separate experiments process the grey scale images directly, using low-level IP primitives and scale-space filters.

[Esparcia-Alcazar & Sharman, 1996] use GP to evolve filters for three Digital Signal Processing (DSP) applications: channel equalisation, noise cancellation, and interference removal. Using a hybrid GP with simulated annealing approach, filters were evolved in each case which performed at least as well as one produced by a Recursive Least Squares (RLS) method.

Ordinary un-typed GP is mainly suited to classification tasks, where the return value of the program can be interpreted as a response to a particular feature. In these areas it can be used as a localised algorithm, concentrating on a single pixel, or small neighbourhood of pixels, at a time. None of the work mentioned in this section uses strong typing, so representations are invariably at the pixel-level or involve pre-processed feature vectors which GP must classify. A particular shortcoming of most of the work is that access to the signal data is restricted to local areas of the signal. This simplifies primitive set design but ignores the potential uses of global or regional signal properties. Notable exceptions here are [Teller & Veloso, 1995], [Poli, 1996] and [Isaka, 1997] each of which provide ways to access arbitrary portions of the signal data.

2.9 Current issues in GP

GP shows a lot of promise as a generally applicable machine learning technique. The power of the representation and the suitability of the computational approach to problem solving indicate that GP can address an enormous range of problems. As it stands currently, however, GP has a hard time solving large problems and suffers from many of the problems associated with genetic algorithms, whilst techniques used in GAs cannot be guaranteed to be applicable to GP. A lack of theoretical understanding of the search processes used in GP hamper its development into a more powerful method. This section mentions many of the issues in GP research that need to be addressed if GP is to exceed its current limitations.
2.9.1 The Building Block Hypothesis in GP

A common explanation of why the genetic algorithm works so well is the Building Block Hypothesis (BBH) [Goldberg, 1989]. This uses the concept of building blocks, which are short schemata that contribute highly to the fitness of individuals containing them. The propagation of highly fit building blocks, which comes from the selection process, and the mixing of these blocks together via the genetic operators, is thought to explain how the GA arrives at a good solution. Each building block develops independently and when several such blocks occur in a single individual, that individual is very fit and propagates accordingly. Such a hypothesis relies on a correlation between occurrence of the building block and a high fitness value, for selection to cause the building block to propagate.

This correlation depends on the extent to which particular alleles at various points in the genome interact. This phenomenon is called epistasis, and is a functional linkage between the alleles present at different points in the genome. Ideally there would be no epistasis at all, and each section of a genome would confer fitness regardless of the values present elsewhere. In this case many building blocks could develop independently and favour individuals where they were combined by endowing those individuals with greater fitness than those with just a single block. Then new, longer, building blocks would emerge and eventually be combined with other blocks to produce an optimal offspring. With an epistatic landscape, however, the presence of a building block in an individual is not a guarantee of improved fitness. The presence of the block is only beneficial if another block is present with a particular form, i.e. the schema required to guarantee improved fitness is much longer, possibly non-contiguous, and therefore much less likely to occur.

Epistasis is almost impossible to avoid in GP. The presence of a particular subtree of code does not carry a guarantee of improved fitness, unless it is in exactly the right position and its computation is used in the right way by all of the nodes from the root of the subtree up to the root node. Every subtree is inextricably linked to every other part of the program tree as values are manipulated in a bottom-up fashion. Only trees with side effects, such as modifying global memory locations, can attain any real form of position independence. For such branches, it is the order of these side effects that matters, and this is still linked to their position in the tree because of the implicit execution sequence present. Additionally, many branches of code are simply not executed, due to conditional statements above them that divert execution along another path, or are executed then ignored because of little pieces of useless code such as multiplying by zero, which will
always return the same answer. This means that the presence of the subtree is not guaranteed to improve fitness as it is always position and context dependent. In certain cases, the primitive set can be designed such that building blocks can attain this independence of position but this is not always easy. The ‘chaining’ method developed in Chapter 6 is an example of such a primitive set.

The BBH relies upon a schema theorem for a definition of a building block. Further problems thus arise from the fact that for GP the definition of a schema in a program tree is not a simple one. In GAs the representation is a fixed length string and a schema is defined in terms of the values present at each point along this string. Since the shape of a GP program tree can vary, and so can its size, many definitions of what constitutes a schema have been produced ([Koza, 1992], [O'Reilly, 1995], [Poli & Langdon, 1997a]). A detailed criticism of the BBH, particularly for GP, can be found in [O'Reilly, 1995, Chapter 4].

2.9.2 Producing better solutions - scalability and generalisation

Often GP will solve particular instances of a problem with little difficulty. These instances may be test cases used within an experimental run. The solutions evolved may well fail, however, when it comes to tackling other (possibly more difficult) instances of the same problem.

There are two issues involved with the tendency of GP to produce solutions that only solve some instances within the problem domain. The first is generalisation, where a solution may have evolved which is tailored to the fitness cases presented during training and that cannot cope with other cases that do not have similar properties. Work on applying GP to eye detection, for example, produces good detectors on the training set but produces high false-positive rates on out-of-sample data [Robinson & McIlroy, 1995].

The second issue is one of scaling, where GP (or a solution produced by) can solve problems of a certain size but not larger instances. Many of the techniques mentioned in this chapter, particularly those devoted to modularisation, are aimed at increasing the scaling properties of GP. The ‘size’ of a problem is not a property that is easily defined. It can be thought of more as a difficulty level for the GP search, as the requirements for a good solution get more stringent or the search space larger. An example would be the Boolean parity problems, which require GP to evolve an expression that correctly reports the parity status of a set of Boolean variables. A problem with 6
variables would be easier to solve than one with 11 variables, thus the size of the set of variables is an indicator of the difficulty of the problem. In other cases it is the configuration of the instance, rather than some easy measure of size, that makes for a more difficult problem.

Generalisation properties allow solutions produced by GP to solve instances of the problem that were not in the training set used. This out of sample testing is a good test of the quality of the solutions and is commonly used in other machine learning disciplines. To improve generalisation, often many fitness cases are used and performance is taken as some function of the set of performances for each individual over the range of fitness cases. This helps to avoid over-fitting, where a solution becomes highly adept at solving the fitness cases used in training by exploiting some unintended property present in that training set. This solution is then useless when used on other, out of sample, cases. The aim of learning techniques is to discover a model of the underlying process from a sample of training data, when exhaustive evaluation of all possible data is impractical and/or impossible.

In addition to increasing the sample size of fitness cases, it is possible to use different fitness cases for each generation, perhaps drawing randomly from a pool of cases, to avoid presenting too static a target to the search. This can make search more difficult, however, as promising solutions may not be rewarded sufficiently in the face of a moving target. It has been observed that smaller solutions exhibit better generalisation properties than large solutions [Kinnear, 1993]. This is thought to be due to the increased chance of a small program having captured some abstraction of the problem domain which saves time and space in the program.

Other methods of attacking the scalability issue include co-evolution, and sliding fitness functions. A sliding fitness function starts off by presenting relatively easy fitness cases to the population, to encourage the production of low-grade solutions. As these begin to dominate, the fitness function introduces harder cases to weed out the weaker partial solutions present in the population. By 'raising the bar' constantly, the search is forced to find solutions that have better solving power, and is less likely to converge on a poorer-performing area of the search space. The method of co-evolution is an example of an adaptive sliding fitness function. The technique of Limited Error Fitness (LEF) [Gathercole & Ross, 1997] is another method of altering the fitness function to make the problem progressively harder.
Despite these advances, scaling remains a difficult issue and one that deserves serious attention. One criticism of GP is that it can only solve 'toy' problems and remains of little use in the real world. By adapting GP to solve progressively more difficult problems, the technique can become much more widely applicable.

2.9.3 Evolvability
The need to balance the exploitation of good solutions with exploration of the search space is at the heart of genetic search. Although exploitation offers the possibility of better solutions than found so far, and is likely to give a short term gain in performance, if not balanced by a more speculative search the run is likely to grind to a halt as it clusters in one area of the search space. Conversely, the wild nature of genetic operators means that randomly stumbling upon a single-point optimum is extremely unlikely, so some form of exploitation is needed once a promising avenue is discovered.

Balancing these two forces is the key to good evolvability. Once a niche is found that deserves exploration, such exploration should be thorough. If the population converges too quickly, possibly fruitful opportunities for exploration of novel solutions are lost. As yet there is no global method that will ensure good evolvability - it depends on the interactions between the genetic operators, the selection methods, and the fitness function. Since this is problem-specific, it is unlikely that one method will ever be universally applicable, so a combination of methods is used instead.

2.9.4 Amenity to analysis of GP
GP remains very hard to analyse and reason about, even more so than ordinary genetic algorithms. The traditional tools of GAs, such as the Schema Theorem and Building Block Hypothesis, have not been successfully translated even to canonical GP. It is not clear how work on landscape analysis applies to GP either. For this reason, techniques in GP tend to have empirical support rather than formal support. Often methods have been introduced or justified on the basis that they improve performance for a particular problem domain, and then left at that (e.g. [Haynes et al., 1995]). While this doesn’t necessarily mean that the techniques are limited in their applicability, it leaves researchers with little clear idea as to whether such techniques will help them in their work. A notable exception to this is the use of ADFs which has been adopted by many researchers thanks to Koza’s demonstrations of improved performance over a wide range of problem domains.
2.9.5 Iteration, recursion and the halting problem

The ability of a genetic program to express the kind of computation required to solve a problem depends on the power of the programmatic constructs available in the primitive set. Powerful methods such as iteration and recursion could well enhance the power of GP as a problem solving method, but they bring their own problems. Unless extremely constrained, iteration and recursion introduce the uncertainty of the Halting Problem into genetic programs.

Approaches to this problem vary. One way is to constrain the recursive function or the iterative primitive so that it is guaranteed to terminate. Another approach is to restrict the number of recursive calls by restricting the depth of the program stack, returning a null value if the stack overflows. Others simply allow the programs to run as they wish and terminate them after a pre-set time, taking the contents of a memory cell as the answer [Teller, 1995]. See section 2.6.2 for other work in this area.

2.9.6 Implementation restrictions - tree sizes, growth, and parsimony

The ways in which genetic search are carried out can often be restricted by implementation details. In particular, program growth is seen as a large problem. It seems to be an unavoidable fact of GP research that the average size of a program tree increases as a run progresses. This places greater memory and computational strain on the system running the search, and is inconvenient.

This phenomenon, known as ‘bloat’, comes about because larger programs are more often fitter than smaller ones. [Langdon & Poli, 1997] argue that bloat is an inherent consequence of simple static fitness-based selection, as a fit individual or schema can occur in a greater number of long individuals than short ones, thus increasing the chances that a long individual will be the one to be selected.

In untyped GP, a maximum tree size is normally imposed by implementations. The traditional processes of program tree construction and mutation sub-tree construction are designed to choose terminals only when the tree reaches a certain depth, and the genetic operators fail (producing identical offspring to parents) if the resulting trees would violate maximum depth restrictions. This is a hard limit, often depth based, so the size of the trees depends on the fanout of the primitives available. In STGP this restriction is not always possible, as it cannot be guaranteed that a terminal exists (or will be selected sufficiently often if it does) for every type present in the system. In such cases, either the primitive set
needs to be modified or alternative methods of restricting tree sizes are employed (see Appendix A).

This upper limit on size is often seen as insufficient, however, so other techniques are employed to restrict the way in which trees grow. Given that longer trees tend to be fitter [Langdon & Poli, 1997], it is not possible to restrict program growth through selection methods or genetic operators alone. Indeed the crossover operator, the most widely used in GP, will not increase the average tree size in a population since it neither creates nor destroys program nodes. Selection methods will increase average program size if longer programs enjoy a fitness advantage (for an exception to this see [Ryan, 1994] and section 2.4.2). Control of program growth tends to be through fitness-based methods, therefore. Usually some kind of parsimony element is included in the fitness function, penalising overlong programs and favouring shorter ones. This element has to be carefully balanced against the other elements of the fitness function, otherwise the population becomes too small and good solutions are unlikely to be found. Approaches to parsimony vary. [Zhang & Mühlenbein, 1995] present a generic framework for balancing parsimony and accuracy (fitness) in genetic programming problems. [Kinnear, 1993] uses a fixed ‘size factor’ in the fitness function when evolving sorting trees to penalise longer trees. [Koza, 1992, section 25.12] finds that adding a parsimony term to the fitness function for a boolean multiplexer term significantly increases the number of individuals that must be evaluated to produce a 100% correct answer. [Iba, de Garis & Sato, 1994] use a Minimum Description Length (MDL) principle to assess decision tree genomes for a boolean concept learning problem. Trees evolved with the MDL principle were more compact and produced better solutions that trees evolved without using MDL.
Chapter 3

GP for Detection Problems

This thesis is concerned with using GP to solve detection problems. This chapter details the ways in which detectors can be evolved and how the evolution of such detectors needs to be carefully managed. Taking inspiration in particular from the work of Canny [Canny, 1986] (see section 4.1.1 for more detail on this), this chapter discusses in a general manner the implications of using GP to solve detection problems. In particular the issues of how to structure detection functions and rate suitable detectors are addressed in a generic way, independent of particular problem domains. A generic fitness function design is proposed that works to minimise misclassification errors in detection problems. This fitness function design is used in the practical work in the next four chapters of this thesis. The use of strong typing and its effect on fitness function complexity is discussed, and is illustrated by the work on template matching in Chapters 5 and 6.

3.1 The nature of detection problems

The problems in this thesis are detection problems, they discriminate data containing desired features from all other data e.g. an edge detector discriminates ‘edge’ from ‘non-edge’. This binary discrimination is in contrast to recognition problems (such as face recognition), which discriminate particular cases within a single class of data. A further class of problems are classification problems (object recognition being an obvious example) where multiple classes of data are discriminated. Recognition and classification problems are usually much more difficult than detection problems in the same domain, and we expect would require much greater effort on the part of the designer and the GP
Detectors take a piece of data as input and apply some algorithm to produce a response. This algorithm normally consists of some kind of transformation of the input data, ultimately followed by a final decision process to produce a binary output. Using the edge detection example from Chapter 4, the input consists of a 1-dimensional signal. It is transformed by convolution with a filter followed by non-maximal suppression, and a thresholding process then decides what is an edge and what isn’t. The output is a binary signal containing decisions for each pixel of the input signal. The decision process is often tuneable with a parameter to adjust the sensitivity of the algorithm, in this case the value of the threshold determines how strong an edge has to be before it is flagged as such. If the decision process is to be optimised, it may be that the parameters are sufficiently small in number that other optimisation techniques such as simulated annealing, or even exhaustive search, would be a better choice.

With GP both stages of the detection process may be optimised, or only one stage. Depending on the problem, it may be more convenient to keep the decision process fixed and just to optimise the transformation. This is the approach taken in Chapter 4, where the optimisation occurs on the shape of the filter used to convolve the input signal. [Poli, 1996] also uses this approach, optimising the filter to transform a picture such that a fixed threshold of the output produces the correct result. Work in Chapter 7 allows both stages of the process to be optimised simultaneously.

### 3.2 Desirable characteristics of a detector

Regardless of the particular detection problem we are addressing, there are some generic properties we can reward in a detector, particularly in any initial signal transformation. These properties correspond to the ones outlined by Canny [Canny, 1986] and are used to drive the edge detector work in Chapter 4.

Firstly, the detector should give a strong response to input that is a positive example of the feature to be found. Ideally the strength of the response in the signal transformation stage would be a monotonic function of the quality of the input data as a positive case, a good match provoking a large response. This would allow the subsequent binary decision process to discriminate more clearly between feature and non-feature data. This property is known as response strength, and if noise is present can also be thought of as the signal-to-noise ratio (SNR) of the detector.
Secondly, when the detector responds to a positive example, it should do so in as timely a fashion as possible. This means that the output of a strong response should be highly correlated in time and/or space with the presence of the feature in the detector’s input field. This rewards accurate detectors with good localisation properties, essential since many detection problems involve an element of search to locate the feature in addition to highlighting its presence.

Finally, the detector should produce no response, or very weak responses, when the input contains non-matching data. This goes hand in hand with the first requirement. It may well be possible to have detectors that respond strongly to correct inputs, simply because they respond strongly to any inputs! Thus we need to specify that the detector should only produce strong responses when we want it to, and at no other times. A decision procedure can then be used, based on the outputs of the detector, to discriminate successfully between feature and non-feature data.

There are trade-offs involved in satisfying these criteria. In edge detection filters, for example, good localisation properties mean rapid strong responses when a feature is found, which implies a high bandwidth in the detector. This increases the response of the detector to noise in the signal which is typically high frequency and thus reduces the noise suppression performance. Since the criteria are antagonistic, solutions built around them are compromises between these opposing requirements.

The two response criteria correspond closely to Type I and Type II errors in statistics. A strong response to a true match implies reducing the chance of rejecting a positive example, a Type I error. A low response to non-matching data reduces the chance of accepting a negative example, a Type II error. These criteria are designed to maximise the number of True Positive and True Negative responses we get (and thus minimising the number of False Positive and False Negative responses) by separating the clusters of response values to feature and non-feature data as much as possible. The rates of True Positives and False Positives however are related so performance characterisation techniques such as Receiver Operating Characteristic (ROC) curves may be needed to show how well these detectors perform for various detection rates. In the experiments in this thesis the costs associated with each type of mis-classification (for instance Table 5.3) can be altered to suit the requirements of GP, where the competitive environment between individuals, and other components of the fitness function (such as parsimony measures), can make choices of penalties and rewards less obvious.
3.3 Designing fitness functions for detection problems

There is no way of producing a fitness function that is common to all detection problems. There are commonalities between detection problems, however, that can make detection fitness functions easier to construct. All detection functions seek certain patterns within a stream of input data. In many cases, some transformation occurs to produce a form of the data more suitable for analysis. The decision process takes the (possibly transformed) data and produces a positive or negative reply. This decision process can be composed from elements expressing the three desirable properties discussed in section 3.2. For example, the fitness function used in Chapter 4 implements the three criteria explicitly, analysing the output signals for the strongest response, the location of that response, and the degree of responses to false data. This three component structure is a good method for approaching a wide range of detection problems with GP.

3.3.1 Fitness function complexity and its relation to the problem, to the type system and to the chances of success

The chance of finding a solution to a problem depends largely on the search space formed by the fitness function and primitive set (see Chapter 2). When designing such a space, it is necessary to consider a number of issues which can have a large effect on experimental and search performance.

3.3.1.1 The exploitative nature of GP and fitness function design

The nature of genetic search means that any method that increases a program’s fitness, no matter how it works, is likely to result in a proliferation of programs that use that method. The search process exploits any avenue it can find for increased fitness. This means that fitness functions need to be extremely tightly defined, leaving as few ‘holes’ as possible, where a ‘hole’ is a way for a program to generate a high fitness value that was not foreseen by the designer of the function, leading to undesirable solutions.

One example of such a hole would be if the criterion for low responses to non-matching data were not present. In such a case, the remaining two criteria could be trivially satisfied by a program that regardless of the input signal produces an output signal of uniformly high value. This would mean that all the features in the input signal would be guaranteed to have a corresponding high output value (because the entire output signal is high-valued), so programs would be regarded as fit by the first criterion. Further, strong outputs would be closely correlated with the input features by the same
token, so localisation properties would also indicate an extremely fit program. Without
the third criterion to penalise such programs, the search would most likely get stuck on
such a false optimum solution. The three components of the generic fitness function
proposed above form a very good generic detection solution, each criterion helps fill gaps
left by the other two that could be exploited by GP.

3.3.1.2 Fitness function types, complexity and STGP
The methods used to determine the fitness of an individual are generally dependent on the
problem domain being tackled. However, some standard methods of fitness assessment do
exist in untyped GP and can be applied to a wide range of problems. The two main
methods are error-based and hits-based. Error-based methods, for each fitness case used,
know the correct answer for that fitness case. The fitness is measured as the error
(difference) between the ideal answer and the one actually returned by the candidate
program, either averaged or totalled over all fitness cases. This method can be augmented
by squaring the error values to give a greater penalty to unfit individuals, or one of any
number of methods of tuning.

Hits-based methods again know the correct answer for each fitness case, but only
measure error on a binary basis. For a given fitness case, the program either gives the
correct answer or it doesn’t. The fitness of an individual is the number of correct answers
(hits) given over all training cases. This is a coarser approach than the error-based
approach but is more suited to discrete problems where the problems are more oriented
toward classification or some kind of target attainment. For example, in the Santa Fe Trail
problem (also known as the ‘artificial ant’ problem in GP), Koza uses a hits measure to
count the number of pieces of food found on the trail by a particular ant [Koza, 1992,
section 7.2]. Both hit counts and error-based fitness measures are used extensively
throughout the rest of Koza’s book.

These methods of assessing fitness are widely used because they are simple to
implement and are appropriate ways of awarding individuals for small improvements in
fitness. More complicated methods involve either the combination of several factors by
weighted sum, or product, to produce a single value, or use of such techniques as Pareto
optimality [Goldberg, 1989] for fitness functions with multiple objective components,
where individuals are awarded for high performance in any of a number of possibly
unrelated fitness criteria.
A third method of assessment is by some properties of the returned values, which are rated by the fitness function and assigned a score. In particular, time series of results can be assessed in terms of several properties and the ways in which these are combined can be crucial to the performance of the search. This is the approach taken in Chapter 4, where the ideal output signal is not known and therefore we can only deal with the properties of the signals returned. Work in later chapters of this thesis encodes the three criteria both explicitly and implicitly, but use error-based methods such as correlation to rate performance. The three-component approach to fitness function design is suitable for both error-based (and hits based) and property-based fitness functions.

Much of the work in this thesis uses strong typing in the primitive sets, indeed we regard strong typing as an essential tool for a wide range of detection problems. The data returned by strongly typed program trees can be, if desired, very complex. In this situation, fitness function complexity increases dramatically. If an arbitrary data type can be returned, as is the case with STGP, the method used to assess that data can be much more complex than if it were a single number, since the number of different assessments is considerably greater. Assessment of aggregate or dynamic data types means that error-based methods become more complex due to the use of more sophisticated distance measures. Property-based methods can be based on more abstract properties which can be harder to debug and optimise. Combined with the propensity of genetic search to exploit holes in the objective function, this makes the design of the fitness function a much more difficult task. In particular, the work in Chapters 5 and 6, which produces lists of line segments as the returned data from a program tree, could use any number of fitness functions at various levels of representation. Distance metrics using the symbolic representation of lines might be used, for example, instead of the correlation-based measure employed on the rasterisation of those lines. Each choice made when designing the fitness function has its own implications and pitfalls for the designer.

3.4 Machine learning of detectors with Genetic Programming

In producing detection programs with GP, a number of factors need to be considered. In common with all applications of GP, the encoding of the problem and of the solutions must be specified and an appropriate fitness function designed. The parts of a software system which are actually evolved by GP may in fact be very small. Often it is the case that the evolved code is executed in the context of a larger body of code which can

72
provide normalisation, error-checking and recovery, and act as a problem-specific 'caddy'. This code would handle all input/output operations, implement any data types required, and be responsible for maintaining the state of the problem as each individual is assessed. Changing the proportion of code that is evolved, compared to the total code size to implement the problem, means changing the amount of work that needs to be done by GP and by the designer. It is easier, from the point of view of maintaining closure, to evolve a smaller program that has to do a smaller amount of work and to enclose this in a large body of code. This is the approach taken in Chapter 4, where the filter produced by the GP is a very small part of the problem implementation in terms of the amount of code produced to perform the signal manipulations, storage, fitness function and other housekeeping tasks. The fitness function in this work however is very hard to maximise so the problem is still quite difficult.

Also important is the nature of the program evolved as a detector. It is necessary to decide whether to produce a detector that flags the presence of the desired feature, or to produce a transformation of the input data that can be analysed and rated by external code. This decision has a big effect on the types of primitives used and on the fitness function, since analysis of an output signal can be much more complicated when the signal is real-valued as opposed to the binary outputs of a classifier or decision process.

Detection is not usually a straight discrimination between positive and negative examples. In real problems the proposed classification of the output is often accompanied by some form of confidence assessment indicating the strength of match found. This is the case when using fuzzy logic, an expert system, a neural network or any statistical model. In each case the adjustment of the system resulting from a reward or penalty factor can be tempered by a low confidence. Binary discriminations threshold confidence values and return positive results only for those outputs of sufficiently high confidence.

Finally, the performance of the detectors produced by GP should not be assessed solely on the basis of fitness values produced during training. It is necessary for the detectors produced to have generalisation ability, to be able to perform well on examples of data that have not been used in training. Traditional experimental design methodology demands that the makeup of the training set and test set should be representative of the application domain ([Haralick, 1994]), such that they capture the abstract properties of the features to be matched and the processes that gave rise to them, i.e. they construct a good model of those underlying processes. Overfitting occurs when the solutions capture
noise in the training set that is not present in the underlying model, and thus perform badly on out of sample test data drawn from the same distribution. Methods of avoiding overfitting have been discussed in Chapter 2, the most applicable to detection problems being to vary the fitness cases used in each generation. Such a technique was used in early versions of the edge detection work used in Chapter 4, with the added benefit of reducing CPU requirements. Later edge detection work uses separate test and training sets, of such sizes that overfitting becomes extremely unlikely.
Chapter 4

Evolving an Edge Detection Filter

Initial practical work for this thesis concentrates on a classic detection problem in the computer vision literature, that of edge detection. This work aims to show how GP can produce very high-performance solutions to complex optimisation problems. Edge detection is a problem to which a large amount of effort has been devoted in the computer vision literature as it forms an essential first step in many machine vision systems, for example, to extract important image features as the first stage in the signal-to-symbol process. Many segmentation methods rely on finding edges (i.e. boundaries) between objects. Tracking systems often use edges or corners to match areas of an image over successive frames. Detecting edges well is therefore an important problem. In addition to a considerable body of work on applying different optimisation techniques to the detection of edges in images ([Chao & Dhawan, 1994], [Bhandarkar, Zhang & Potter, 1994], [Srinivasan, Byatia & Ong, 1994] are just a few examples), theoretical work on the properties of an ideal edge detector has produced a number of high quality operators, each satisfying slightly different theoretical requirements aimed at optimising the accuracy and reliability of the edge detection process (see section 4.1.1).

In this chapter the theory of edge detection [Canny, 1986] is applied to the automatic production of high-quality detectors. Using GP, we show how detectors can be evolved that are tailored to the data set upon which they are to be used, thus improving overall detection performance. This work appeared in early form as [Harris & Buxton, 1996], and the full set of results was published in [Harris & Buxton, 1997].

Edge detection, like many image processing operations, involves transforming an
image and subsequently analysing the transformed outputs. The transformation performed in edge detection works by highlighting pixels within the image that represent discontinuities in image brightness. This is normally achieved by convolution of the input signal with a digital filter [Gonzalez & Wintz, 1987]. This experiment uses GP to produce and assess filters for use as edge detectors. Section 4.1 summarises the theoretical properties of a good edge detection filter and gives examples of good filters developed from this theory. Section 4.2 shows how GP can use this theory to produce such detectors, followed by details of our experimental work. Section 4.3 presents work showing comparative measures of performance of detectors on various sets of training data. The results are summarised in section 4.4 and some of the best detectors evolved are shown in section 4.5.

4.1 Convolution Filters for edge detection

Edge detection filters are functional forms over a fixed interval, either [-1,1] or [-1,0] (in which case the function is inverted in the origin to guarantee an anti-symmetry property, and thus covers the whole [-1,1] range). These functions are used as convolution kernels to transform an input signal or function, so as to highlight edges, which are discontinuities or sharp changes in the input, by producing corresponding extrema in the output. The extrema are directional, a maximum indicating an edge in one sense, a minimum indicating an edge in the opposite sense. The simplest case of an edge is a step edge, which consists of a transition of infinite gradient in the signal from one steady value to another, as shown in Figure 4.1. The theoretical treatments mentioned in this section assume that all edges are step edges and that the signal is corrupted with Gaussian distributed noise.

4.1.1 The optimal filter. A theoretical treatment of edge detection filters

[Canny, 1986] presented a set of criteria that an ideal edge detection filter would fulfil. Three criteria were presented that contributed to the performance of a filter on noisy edges of various kinds. A good signal to noise ratio would ensure a good 'hit rate' and minimise the number of edges missed by increasing the likelihood of a strong response to an edge. A localisation criterion ensures that the filter responds most strongly to the edge as near as possible to the actual edge location. Finally, a noise suppression measure is incorporated to minimise the number of false responses of the filter to noise in the signal that doesn’t signify a true edge. Canny derived expressions for these three quantities and
combined the first two to produce an expression for an optimal filter for step edges. Numerical optimisation of this expression gave an "ideal" detector which Canny approximated with the first derivative of a Gaussian function.

[Spacek, 1986] took Canny’s work and derived an expression for the optimal operator using all three criteria, with slightly different boundary conditions. Spacek’s work assumed an interval for the detector function of [-1,0] and inverted this function about both axes to guarantee anti-symmetry. The expression he derived for the optimal operator depended on six parameters, the values of which determined the performance of the operator. Spacek fixed two of these and optimised the other four to produce his optimal detector. According to Spacek, this optimal detector outperformed Canny’s ideal detector, using the same measure of performance as in Canny’s work, as well as the Gaussian derivative approximation. A convenient approximation to Spacek’s optimal detector, which also outperformed both of Canny’s operators using Canny’s own criteria, was a cubic spline function, \( x^3 + 2x^2 + x \). This approximation had over 99.9% of the performance of Spacek’s optimum according to his own performance measure.

[Petrou & Kittler, 1988] extended Spacek’s work to optimise all six parameters in the expression for the optimal operator. The resulting form improved on Spacek’s optimal operator and on the cubic spline operator using the same criteria as Spacek. When applied in practice to real images however, the difference in performance was not found to be significant.

The functional forms of the operators derived can be found in Table 4.4. Plots of these functions are shown in Figure 4.2.
4.1.2 Translation to the digital domain

Since we are working in a discrete domain, the symbolic forms of the detector functions need to be sampled over the appropriate interval to produce digital filters. The detectors derived in the theoretical work are optimised to work on edges of infinite gradient, meaning the edges have effectively zero width. This type of edge has the same profile no matter what scale it is viewed at.

In the digital domain, where signals are represented by a discrete series of values, true step edges of infinite gradient cannot be constructed. An edge in a digital signal has a minimum width of a single pixel, so the effective gradient of the edge is always finite. A finite width edge of some shape thus has an associated scale, related to the total width of the edge feature. In machine vision applications edges are often detected over a range of scales to increase the reliability of the detectors.

The lack of true step edges and the notion of edge scale means that the translation to the digital domain inevitably involves some deviation from the theoretical conditions under which the optimal detectors were developed. To compensate for this to some degree, the detector function, which is sampled to produce a filter, can be sampled at higher rates (see Figure 4.3) to apply the filter at different scales. Edges of different scales can thus be detected. Furthermore, this multi-scale approach allows the detector to cover a larger portion of the signal at any one time, increasing the effective gradient of any edge by reducing its width in comparison to the width of the filter. Large filters applied to narrow edges operate under conditions closer to the theoretical because the effective gradient of the edges, relative to the size of the filter, is higher.

Convolution of an input signal with an edge-highlighting filter produces an output signal. In 2-D this convolution may either be with a 2-D filter, or a combination of oriented 1-D filters. The output signal highlights discontinuities in the input. In [Canny, 1986], the convolution is followed by a process of non-maximal suppression, in which all responses that are not maxima in the output are set to zero, to localise the edge pixels. If the signals are 2-D the suppression phase is followed by a final hysteresis phase where proximal edge fragments are linked together into larger structures. The hysteresis process may also, if the signals are analysed at several scales simultaneously, synthesise edge fragments based on evidentiary contributions from the corresponding portions of the output signal at each scale used. The combination of non-maximal suppression and hysteresis implements the decision process separating edge pixels from non-edge pixels.
These two extra stages help to reduce edge fragmentation, and lower the incidence of spurious edge fragments in the image.

4.2 Evolving a filter with Genetic Programming

The change in conditions caused by the translation to the digital domain means that the 'optimal' operators derived in the theoretical work may no longer be truly optimal. Additionally, real-world data contains edges which do not fit the assumptions made about edge shapes and will thus respond less strongly to such detectors. This opens up an opportunity to produce detectors that are optimised for work in the digital domain, and for particular training sets of edge profiles taken from real-world images. This problem, being an optimisation problem with high-performance solutions already known, also provides a good challenge for GP. Our experiments concentrated on the evolution of the functional form used to produce the digital filter in the first stage of the detection process. This filter is used to highlight edge pixels in a series of training signals, and the output signals produced are analysed to produce a performance measure, as detailed in section 4.2.3.

4.2.1 Five preparatory steps

Applying GP to a problem domain involves five stages of preparation: choosing the terminal set, choosing the function set, designing a fitness function, fixing experimental run parameters, and choosing a termination criterion.

The problem is to evolve a filter that, when convolved with a series of 1-dimensional real-valued input signals, will highlight the discontinuities in the input signals with extrema in the output signals produced.

In these experiments, such a filter will be produced by sampling a numerical function, so we need to evolve the numerical function. It is necessary to choose function and terminal sets that allow any numerical function to be expressed by combinations of the primitive elements, whilst not biasing the search in any particular direction. For terminals, we need to include the independent variable \( x \) (the position within the \([-1.1]\) interval) as domain knowledge, and some useful constant values to allow the formation of arithmetic expressions.

Table 4.1 lists the terminals used to construct the program trees, and Table 4.2 lists the functions used. These choices give a large range of possible answers, and satisfy
the sufficiency criterion mentioned in section 2.7.3 by allowing the possibility of Taylor series through expressions involving coefficients of powers of \( x \). The primitive set allows GP to evolve all of the three forms used in the detectors known to perform well under theoretical conditions (see Table 4.4).

The choice of functions, and the special nature of the \( \text{pow} \), \( \text{rlog} \) and division functions in particular, is dictated by the necessity to satisfy the closure property required when using GP. This property requires that the behaviour of all functions in the function set is well defined for all possible values of its parameters. This means that every genetic program that could be produced is still semantically valid, even though the results produced may not be intuitive. We can thus guarantee that any program will execute without causing an error, a necessary precaution when the structure of the program is governed by random factors. The \( \exp \) function is included to see if GP can produce solutions based around a form similar to the first derivative of Gaussian used in [Canny, 1986].
The fitness function is a faithful interpretation of the theoretical requirements for a good detector detailed in [Canny, 1986], and is explained in section 4.2.3. Experimental run parameters can be found in the tableau for the experiment (see Table 4.3). Each run performed was continued for 50 generations, and the overall best-performing individual was taken as the final output of the run. The termination criterion is that the run has completed 50 generations.

### 4.2.2 Experimental design

Each detector function is sampled over the specified interval to give an array of coefficients that constitute the filter. These coefficients are then normalised to ensure that the absolute area under the filter was equal to 1. This ensures that the shape of the detector is what is important, and not any arbitrary scaling that may exist in the genome of the detector. Each detector is sampled separately at a number of different sample rates, to give a collection of operators at different scales. We sampled the detector function 7, 9, 11, 13, 15, 17, 19, 21, 23, 25 and 27 times to give a total of 11 different filters per genetic program. Each operator is then a filter with a different range of support from 7 to 27 pixels. For each sampling rate, samples were always taken for $x = 1$, -1, and 0, with the remainder of the samples being uniformly distributed over the interval.

Producing a collection of operators for each detector function allows us to minimise the chances of overfitting, i.e. of evolving an operator that takes advantage of particular characteristics of the training set to produce distorted performance. It also means that any evolved function will be less likely to fail when generalised. It is quite
The evolved function starts at the x-position of the evolved function. This will range from -1 to +1.

0.0 A constant.
1.0 A constant.
2.0 A constant.

Table 4.1. The set of terminals used by GP.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, - , * , /</td>
<td>Arithmetic operators with protected division. Dividing by 0 returns 0.</td>
</tr>
<tr>
<td>rlog</td>
<td>Protected log function. Takes absolute value of parameter.</td>
</tr>
<tr>
<td>exp</td>
<td>Exponent.</td>
</tr>
<tr>
<td>pow</td>
<td>Protected power function. Raises first parameter to the power of the second parameter. Rounds down fractional (positive or negative) powers of negative numbers, but allows any power of positive numbers.</td>
</tr>
<tr>
<td>sin</td>
<td>Sine function.</td>
</tr>
<tr>
<td>cos</td>
<td>Cosine function.</td>
</tr>
</tbody>
</table>

Table 4.2. The set of functions used by the GP.

likely that if we only used a single sampling rate, GP would evolve a function that performed reasonably only at that sampling rate, and when used at a different rate would perform badly because the operator is a different shape. We want to produce operators that maintain their shape over a range of sampling rates. Another reason for producing a set of operators from each evolved function is that it allows us to explore the change in performance as the relative widths of operator and edge change. As the operators get larger, the discrete signals can get closer to the theoretical conditions of infinite-gradient, zero-width edges. This leads us to expect better relative performance in our experiments from the theoretical detectors as their width increases.

The operators are all tested on the entire training set used, and the performance of the genetic program that produced them is measured as the average score of the whole set of operators over all the sampling rates used, to avoid overfitting. This performance measure is the fitness measure used to drive the GP. 'Final result' performance measures, detailed in section 4.3, come from applying the same fitness criteria to out-of-sample data from the same image ensemble used to produce the training data ('image ensemble' meaning a set of images with statistical properties drawn from a well-defined underlying distribution e.g. having similar noise characteristics [Haralick, 1994]).

We used three different sets of data to evolve three separate sets of detectors. Each set of training data consisted of a number of 1-dimensional signals, extracted as profiles from sets of related images. These signals were marked by hand at the locations of significant edges. The signals resulting from convolution of the training data with a
Objective: To respond strongly and accurately to edges in a signal, and not to respond to non-edge sections.

<table>
<thead>
<tr>
<th>Terminal Set:</th>
<th>x 0.0 1.0 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Set:</td>
<td>+ - * / r log exp sin cos pow</td>
</tr>
<tr>
<td>Fitness Cases:</td>
<td>3 sets of 1-D signals with marked edges, divided into training and test sets</td>
</tr>
<tr>
<td>Raw Fitness:</td>
<td>$S(f)C(f)/L(f)$ - see section 4.2.3</td>
</tr>
<tr>
<td>Standard Fitness:</td>
<td>(Max raw fitness) - raw fitness</td>
</tr>
<tr>
<td>Selection method:</td>
<td>Tournament, size 2</td>
</tr>
<tr>
<td>Population size:</td>
<td>500</td>
</tr>
<tr>
<td>Mutation rate:</td>
<td>10%</td>
</tr>
<tr>
<td>Crossover rate:</td>
<td>80%</td>
</tr>
</tbody>
</table>

Table 4.3. Tableau for edge detection filter problem

candidate filter are analysed at the marked locations by the fitness function. The first set of training data were 1-D profiles drawn from a series of underwater surveillance images. These were characterised by high levels of noise and indistinct broad edges. A total of 126 edges were marked in this training set. The second set came from a series of pictures in an office environment. These pictures were noticeably cleaner and edges were stronger, although still relatively wide. This set of data contained 128 marked edges. The third set of signals used to evolve detectors were synthetic signals, consisting entirely of step-like edges one pixel wide, which were then corrupted with Gaussian-distributed noise.

This last set of signals was designed to most accurately reflect the conditions under which the theoretical operators described in section 4.1 were derived. This set contained 52 edges, reflecting the relative uniformity of step edges in comparison to those from real-world images (fewer edges are needed because the variation between the edges is small).

Parts of the fitness function used are applied around hand-marked edge positions in the input signals, as we are interested primarily in the responses of the detector to an edge feature. In each training set, an equal number of left-handed and right-handed edges were marked (i.e. the number of edges at which signal intensity dropped was equal to the number of edges at which signal intensity rose, when moving positively along the x-axis of the signal), so as not to bias the search to produce detectors optimised for a particular direction of edge. Examples of training data from each image ensemble, and an image from the office ensemble, can be seen in Figure 4.4.

For each set of training data, five experimental runs were performed. Each run
uses a different randomly-generated initial population, and therefore takes a different path through the genetic search space.

### 4.2.3 Evaluating the performance of the evolved detectors

The process of evolving successful edge detectors is driven by the fitness function, which provides the mapping from genome to performance for a candidate solution. This function analyses each output signal, i.e. the result of convolving the training signal with the filter sampled from the candidate solution. Three criteria are applied to the signal to produce an overall score. These criteria are designed to correspond closely to those specified by Canny, with some modifications to render the three criteria independent of each other as explained below.

Around the locations marked as edge positions, criteria measuring response strength and localisation error are applied. The response strength criterion searches for the signal value with the largest magnitude, within a neighbourhood of the marked position equal in size to the width of the operator, and takes this magnitude as the score for that edge. The score for all marked edge positions over the signal is averaged and this is taken as the partial fitness value of the filter, $S(f)$ where $f$ is the operator function.

The distance of this maximum response from the original marked edge position in pixels, after correcting for convolution shrinkage of the output signal (a result of having to place the filter entirely within the signal for convolution - a few pixels at either side of the signal are lost), is taken as a measure of localisation error. The score for all edge positions over the signal is averaged, and this is taken as the second partial fitness value, $L(f)$.

The response strength criterion could have been assessed simply by taking the magnitude of the output signal at the edge location, but this would be affected by the localisation properties of the detector also. A high response value, if not well localised, would be missed by such a simple interpretation of the desired characteristic. We therefore choose to ignore the position of the best response value until it is explicitly considered in the localisation criterion.

The third criterion is not localised to marked edge positions. The operator should respond as little as possible to any noise present in the signal, i.e. we require operators with noise suppression properties. To measure the degree to which the operator responds to noise, following Canny, we look at the separation of peaks in the output signal. Since
Figure 4.4. Example training data

we are using the same signals as input for all the operators, the noise suppression performance of an operator is directly illustrated by the average peak separation of the output signals produced by convolving the input signal set with that operator. Better noise suppressors will produce more widely spaced peaks. This value is averaged over all the output signals and taken as the partial fitness value, $C(f)$.

Once the partial fitness values for the operator have been calculated, the final fitness of each genetic program, following Spacek, is calculated as $S(f)C(f)/L(f)$. This means we are looking for operators with high response strength, small localisation error, and good noise suppression. The objective of the genetic search is thus to maximise this term. Since we are only interested in relative performances, it is not important that the value of the response strength score depends to a large degree on the values in the input signal. Using the output signal to rate each detector’s noise suppression performance means the search is tailored to the noise characteristics of the training set. A detector may, if desired, trade off noise suppression performance for gains in another of the partial fitness values, if further minor improvements to noise suppression performance turn out to be of little overall benefit.
4.3 Results and performance evaluation of detectors

4.3.1 Operator groups
After each set of operators had been evolved, one set for each set of training data detailed in section 4.2, the best five performers were taken from each set. This gave fifteen evolved operators in three groups. We then compared the performance of these operators against a group of four of the operators derived in the theoretical work. The theoretical operators used are shown in Table 4.4. This gives four groups of operators (three groups of five evolved operators and one group of four theoretically derived operators), the relative performances of which are then tested on set of out-of-sample data, drawn from the same image ensembles used to produce the training data. The three evolved groups of operators will be referred to as the underwater, office, and synthetic groups, reflecting the training data used to evolve the operators. Details of some of the best operators that were evolved are shown in section 4.5.

4.3.2 Performance evaluation
All four groups of operators were tested using the same fitness function as during training, to measure their performance on out-of-sample instances of data, for each of the three types of training data. The test sets were of similar size to the training data used. This allows us to see how well detectors evolved on a particular set of data perform on new data with similar characteristics, which is what a vision engineer would be seeking in practice, and gives some indication of performance against detectors optimised for different situations, i.e. it shows how specific the operators that we evolved are to a particular image ensemble, and whether we can expect a performance gain by developing detectors specifically for an ensemble.

Each set of results (see section 4.3.3) shows the average performance of all the operators in each group, over the range of sampling rates, for a particular set of test data. These results are then further averaged over all sampling rates to give an overall performance figure. This averaging is in order to gain some appreciation of the consistency of the evolved detectors, rather than look at a single detector which will have individual strengths and weaknesses.

In addition, the number of ‘hits’ for each set are marked. A ‘set hit’ is scored for a particular set if, for a given sampling rate, that set scores higher overall than all other sets in the experiment. ‘Individual’ hits are also given if the best performing individual detector
First derivative of Gaussian as used in [Canny].

\[- \frac{x}{\sigma^2} e^{\frac{-x^2}{2\sigma^2}}\]

Cubic spline approximation, used in [Spacek]. Positive \( x \) values are mirrors of the negative values, to gain anti-symmetry (a necessary precondition in the theory).

\[x^3 + 2x^2 + x\]

'Optimal' operator derived by Spacek.

\[
\begin{align*}
C_1 &= -13.3816 \\
C_2 &= 2.7953 \\
C_3 &= 0.0542 \\
C_4 &= -3.7953
\end{align*}
\]

\[
\left( C_1 \sin x + C_2 \cos x \right) e^x + \\
\left( C_3 \sin x + C_4 \cos x \right) e^{-x} + 1
\]

Optimal operator derived in [Petrou & Kittler, 1988].

\[
\begin{align*}
A &= 1.44609 \\
K_1 &= -5.22049 \\
K_2 &= -0.88171 \\
K_3 &= 0.6 \\
K_4 &= -1.0 \\
K_5 &= 1.86171
\end{align*}
\]

\[
\begin{align*}
\left( K_1 \sin Ax + K_2 \cos Ax \right) e^{Ax} + \\
\left( K_3 \sin Ax + K_4 \cos Ax \right) e^{-Ax} + K_5
\end{align*}
\]

Table 4.4: Theoretical operators used in performance evaluation.

for a given sampling rate belongs to that set. The two hit markings help to indicate the consistency of the performance of a set of detectors, giving some measure of how likely an evolved detector is to perform better than an alternative, and thus how likely GP is to produce an effective solution to the problem.

4.3.3 Results

The results for each set of test data are shown in Table 4.5, Table 4.6 and Table 4.7. Each value in the tables represents the averaged score of the group of operators over all edges in the test set signals as calculated by \( S(f)C(f)/L(f) \). Actual values are unimportant since they depend on the intensities present in the signals themselves. More important are the relative performances between operator groups.

4.3.3.1 Results for underwater test data

At each sampling rate, the “underwater” operators (i.e. those evolved on the underwater image training data) consistently outperform the operators in the other three sets. The underwater set score the highest mark at every sampling rate as a set, and also contain the highest scoring individual detector. On inspection, improved noise suppression performance proved to be the largest contributor to the increased performance of these detectors, as would be expected from the characteristics of the training data (see Figure 4.4(b)). The overall scores are shown in Figure 4.5.

4.3.3.2 Results for office environment test data

Here, the office detectors score best on overall performance. The picture is less clear, however, when the number of hits is considered. At lower sampling rates, the office
detector set enjoys a clear advantage. As the sampling rates increase, the theoretical detectors improve relative to the other detectors and perform best at rates of 17 and above. The best individual performer most often turned out to be a detector evolved on the office data, with occasional strong performances from synthetic or theoretical operators. These results are due to the range of performances present within the office set of detectors, where we found one detector performing consistently worse than the other four in the set. This brings the average score for the set down, whilst still allowing for individual strong performances from other detectors in the set.

The strong performance of the theoretical detectors may be partly attributed to
two factors. Firstly, the training data characteristics (see Figure 4.4(c)) are quite well suited to these detectors, having a low noise component and relatively sharp edges when compared to the underwater set. Secondly, the expected increase in performance of the theoretical detectors as sampling rate increases (see section 4.2.2) could be a factor. The increased performance of the synthetic detectors at higher sampling rates would tend to support this idea.

The relatively poor performance of the underwater detectors reflects the low noise level present in the data. Improvements in localisation and response strength prove to be more important with this data, whereas these were subjugated to noise suppression performance in the underwater detector set. The overall scores are shown in Figure 4.6.

4.3.3.3 Results for synthetic step edge data

The results for synthetic step edge data show some dominance by the synthetic detectors when looking at overall performance. The synthetic operator set scores best overall, and at 6 of the 11 sampling rates. However these are distributed across the whole range of sampling rates and it is difficult to identify clear trends. The results are more clear-cut when looking at individual performances. At 10 of the 11 sampling rates, a synthetic operator scored best. The theoretical operators, as expected, score better at higher sampling rates, and begin to outperform the synthetic operator set at the highest rates. The underwater and office operators tended to score best at the lowest sampling rates, when the theoretical and synthetic operators were expected to suffer in comparison, and drop in relative performance as the sampling rate goes up. The overall scores are shown in Figure 4.7.

4.4 Discussion

We have presented the application of Genetic Programming to the production of low-level edge detection filters. The use of GP allows the evolution of such detectors in symbolic form, permitting their use at any scale desired when put into practice. The evolution is driven by a fitness function directly inspired by the three theoretical criteria developed by Canny and Spacek, describing the properties of a good edge detector.

Results show that, in real-world situations, the theoretical detectors that are derived directly from these criteria do not necessarily have optimal performance. The conditions under which these detectors were developed do not correspond to the
conditions under which most edge detection is performed in practical machine vision applications, because the models assumed in their derivations that edges would be of infinite gradient [Jain & Binford, 1991]. In the majority of cases, a detector evolved for a specific set of data will outperform the theoretical detectors by the same criteria used to develop these detectors, and will outperform other detectors evolved on different training data. This technique allows a data-driven approach to creating customised edge detection operators for use within a specific application. Such detectors have improved performance and high specificity, reflecting the characteristics of the data set used to drive their evolution.

4.5 Evolved edge detection operators

This short section details the genetic makeup of some of the operators evolved during experimental runs. This section gives details of the form of one operator from each of the three operator groups used, as examples of the kind of program trees that are produced.

The operators are represented as trees, with internal nodes being numerical functions drawn from the primitive set specified in section 4.2, and leaf nodes drawn from the terminal set. Each tree is a function in a global value \( x \), and returns a real-valued number each time it is called. In our implementation the genetic programs are represented directly as C program code, and \( x \) is a global variable that is altered each time a value of \( f(x) \) is required. The filters are produced by looping over a number of values of \( x \), calling
the genetic program once each time and filling the filter with the return values of that
function.

Each operator is presented in tree form. It is not intended that these forms be
readily understandable to the reader, indeed GP usually creates solutions in ways that to
humans appear illogical and wasteful, often full of useless code. Nevertheless these
programs are illustrative of the sort of detector functions that can be produced
automatically. Alongside the program trees, plots of the operator are shown in identical
form to those shown for the theoretical operators in Figure 4.2. The operators are shown
in Figure 4.8, Figure 4.9 and Figure 4.10.
Figure 4.5. Edge detector performances on underwater test data

Figure 4.6. Edge detector performances on office environment test data
Figure 4.7. Edge detector performances on synthetic step edge data

Figure 4.8. Operator evolved on training data consisting of synthetic step edges corrupted with Gaussian noise.
Figure 4.9. Operator evolved on training data taken from underwater image ensemble
Figure 4.10. Operator evolved on training data taken from office environment image ensemble.
Chapter 5

Template Matching

The use of GP in Chapter 4 to produce an edge detection filter was an example of how detectors can be evolved in one dimension, although the detectors could then be used in two dimensional applications by applying the 1-D filters horizontally and vertically to produce an edge-vector image [Canny, 1986]. This chapter extends GP to a two dimensional problem and evolves detectors in two dimensions, using a higher-level representation of image features and employing GP to model a feature with a symbolic description. Template matching is a method for locating particular features in an image by generating a template that has characteristics similar to those of the feature to be matched. This template is then used as a model against which features in the target image can be compared [Pratt, 1991]. We use GP to generate and test templates to match line drawings in a raster image.

5.1 Representing an image

Before we can decide on an algorithm by which we are going to detect features in an image, we need to look at the way the data is encoded in the image. The image representation will limit (see Chapter 2) the ways in which we can apply GP to the detection of features.

Images can be represented at many different levels, depending on the information available to the experimenter. Each level of representation encodes information at a certain abstraction from the raw signal input from whatever device is used to produce the image. At the lowest level, raster images encode the image as a matrix of samples of
image intensity (or several intensities in the case of colour or other multi-spectrum images). Little relevant information is available at this level. A higher level of description is segmentation based - the image is divided into areas that are thought to be related in some way, such as belonging to the same object in the scene. Descriptions at this level can include occupancy hierarchies such as quad-trees, or boundary descriptions e.g. chain codes. Higher still are geometric representations that describe the image symbolically in terms of primitive shapes and positional and orientation information. This is better suited to artificially constructed data and can be extremely hard to produce from lower level representations. Other models include relational representations, where objects or regions are related to others by such properties as enclosure, intersection, displacement etc. These representations often use graph methods as an encoding, and allow some kind of reasoning about the image at a symbolic level. A discussion of data structures for image analysis can be found in [Sonka, Hlavac & Boyle, 1993, Chapter 3].

Matching methods vary according to representation, so it is necessary to pick a representation appropriate to the matching method, which in our case is going to be template matching. We choose to use two representations for the experiments in the next two chapters. The first is the basic, low-level raster representation using a single number per pixel to represent intensity values. Matching is done on this low-level representation. The second representation used is a line-based description, which can be formed either from synthetic data or by simple processing and approximation of a raster image to produce a list of line segments. To simplify things further, we restrict ourselves to black and white images which makes similarity metrics easier (since we only have a binary decision at each stage rather than some graded response). In classic computer vision techniques these black and white images could be the output of an edge detector, to which lines would be fitted and then matched to a model database as part of an object recognition system, say.

5.2 Matching images

For a raster image, matching a template image with a section of a larger target image involves computing some measure of similarity for the template and each region of the target. This means the matching process must compare the template with all possible subsections from the target image, and obtain a single measure of similarity for every position, by placing the template over the target at a particular location and assessing
some function based on the values of each pixel in the template and corresponding pixels on the target. This gives an array of similarity measures, which can be used to drive the fitness function.

A simple matching method using raster images is correlation. Simply, the match between a template and a region of a target image is a function of the number of (assuming binary images) pixels in the template which have equal value to the corresponding pixels in the target region. Each pixel that matches increases the strength of the correlation, each mismatched pixel can either be ignored or can penalise the correlation, depending on the requirements of the system.

Such a method can be very successful but is very brittle in the face of small transformations of either the image or the template. In such cases correlation scores can drop off rapidly even with small deviations from the ideal, adversely affecting recognition performance. Computational power is also an issue. Each pixel of the template needs to be compared against a pixel of the target for each location of the template. For a 50 pixel-square template, on a 400 pixel-square image, this equates to

$$50^2 \times (400 - 50)^2 = 3 \times 10^8$$

pixel comparisons for a single image. This load can of course be cut down by using smaller images and templates, at some loss of flexibility. The computational requirements are made worse by the prospect of processing some 500 possible templates (a typical size for a GP experiment) for a single generation, even before considering runs of 50-100 generations and training sets of several tens of images. Some saving could be made by guiding the template to promising areas by using sub-sampled images and templates to produce a scale space [Koenderink, 1984].

To save time we assume that registration of the template and target image has been achieved [Pratt, 1991, section 20.4]. In our experiments we use identically sized images for both template and target, avoiding the need for registration and concentrating on producing the best match for the target data. We use entirely synthetic images as test data, consisting of straight line segments only.

5.3 Generating templates with GP

To apply GP to the problem of template matching, we need to get GP to produce templates in some constrained random fashion which can then be assessed by a matching
criterion. Taking into account the factors mentioned in Chapter 2 concerning completeness of a primitive set, we need to find a way of constructing templates that will allow us to produce an arbitrary binary image from a symbolic description. This will give us the generality required to match any image we care to use as training data.

Since we need to match our templates with a bitmap image, we need some way of translating a symbolic description of an image into a bitmap. In computer graphics terms, we need to render our description. Our primitive set therefore must produce programs for drawing arbitrary graphics on a 2-D canvas. An obvious way to do this is to provide a primitive set that draws geometric primitives in the image. Terminals in the set would be parameter values for functions to draw specific kinds of shapes, and the function set would consist of the shapes themselves plus transformations on the shapes produced lower down in the tree. This would allow the production of a genetic program that could produce just about any binary image, and allows great flexibility on the part of the search.

To encourage the production of images with some apparent structure, it was initially intended to use primitives that encode shapes directly, such as squares, curves, ellipses and lines. When tested in early experiments, however, results were disappointing as it proved extremely difficult to use these high-level shapes and still get good fitnesses. This tended to select out the shape primitives and use solutions based solely on lines instead. In retrospect it can be seen that a complex shape could not be expected to get even partial matches without most of the parameters being exactly correct, which is extremely unlikely. A program using only straight line segments as drawing primitives has the same generality and leads to much better progression as more complex models are evolved, so this was used instead.

The full primitive set used in experiments can be found in Table 5.1. The use of symbolic shape descriptions allows the manipulation of the symbols by transformation operators within the tree, then a single rendering operation on the line segments produced takes place after the tree has been evaluated. The set of affine transformations is included to allow the possibility of exploiting any structural regularity in the images - by copying a grouping of line segments, and applying a transformation, repeated shapes can be conveniently captured by the combination of duplication (via selection) and crossover of subtrees. The inclusion of the transformations, as well as making this exploitation of structure easier, allows for shapes to be constructed without necessarily having to evolve exactly the right constants for the endpoints of each line. A line could be constructed and
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Return type</th>
<th>Parameter types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus</td>
<td>double</td>
<td>double, double</td>
<td>arithmetic addition</td>
</tr>
<tr>
<td>minus</td>
<td>double</td>
<td>double, double</td>
<td>arithmetic subtraction</td>
</tr>
<tr>
<td>times</td>
<td>double</td>
<td>double, double</td>
<td>arithmetic multiplication</td>
</tr>
<tr>
<td>divide</td>
<td>double</td>
<td>double, double</td>
<td>protected arithmetic division</td>
</tr>
<tr>
<td>0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 0.33, 0.25</td>
<td>double</td>
<td>none</td>
<td>constants</td>
</tr>
<tr>
<td>Join</td>
<td>LineList</td>
<td>LineList, LineList</td>
<td>Joins one LineList to another and returns their union</td>
</tr>
<tr>
<td>Rotate</td>
<td>LineList</td>
<td>LineList, double</td>
<td>Rotate a LineList by some amount (in radians)</td>
</tr>
<tr>
<td>XShear</td>
<td>LineList</td>
<td>LineList, double</td>
<td>Shear a LineList in the x direction by some factor</td>
</tr>
<tr>
<td>YShear</td>
<td>LineList</td>
<td>LineList, double</td>
<td>Shear a LineList in the y direction by some factor</td>
</tr>
<tr>
<td>Translate</td>
<td>LineList</td>
<td>LineList, double, double</td>
<td>Move a LineList around</td>
</tr>
<tr>
<td>Scale</td>
<td>LineList</td>
<td>LineList, double</td>
<td>Enlarge a LineList by some factor, around its centre</td>
</tr>
<tr>
<td>line</td>
<td>LineList</td>
<td>double, double, double</td>
<td>Construct a LineList from start and end co-ordinates</td>
</tr>
<tr>
<td>NULLLineList</td>
<td>LineList</td>
<td>none</td>
<td>A terminator used for the Join primitive to allow trees to stop growing</td>
</tr>
</tbody>
</table>

Table 5.1. Primitive set for template matching problem

stretched, for instance, to get a better fit on the image with fewer constants to optimise. The arithmetic operators are included to allow some flexibility of constant values.

The primitive set chosen is strongly typed. It is a property of image processing problems that a diverse set of data types is often used, and it is no surprise that different data types should be useful when applying GP to solve such problems. In this case, it would be extremely difficult to find a primitive set that could produce a structured two-dimensional image from a program consisting entirely of functions that return single numbers. One way in which this might be envisaged is to encode x and y co-ordinates within the canvas as terminals and use GP to evolve an expression which returns a positive number for pixels that should be white, and 0 or negative number for black pixels. However this would be extremely unlikely to produce anything useful considering the possible complexity of the images. This problem illustrates the kind of situations where strong typing is not only desirable but, we believe, essential. Chapter 6 shows how strong
Objective: To produce an image matching the training image with a set of line segments.

Terminal Set: see Table 5.1
Function Set: see Table 5.1
Return type required: LineList
Fitness Cases: 2 experiments, each consisting of one 50x50 pixel image
Raw Fitness: (no. of correct ‘on’ pixels - no. of incorrect ‘on’ pixels) - parsimony constraint
Standard Fitness: (Max raw fitness) - raw fitness
Selection method: Tournament, size 2
Population size: 1000
No. generations: 100
Mutation rate: 20%
Crossover rate: 70%
Parsimony factor: 0.01

<table>
<thead>
<tr>
<th>Table 5.2. Tableau for the template matching problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>typing can be beneficial to program evolution when applied carefully and with consideration of the type of solutions required.</td>
</tr>
</tbody>
</table>

5.4 Experimental design

The experiments in this chapter use GP to evolve binary images that match a target image. In doing so, the solutions evolved are actually symbolic representations that map to a particular raster image when rendered. This means that as well as searching for a good match for a particular target image we are also converting from raw data into a higher-level description.

Using the primitive set described earlier, lists of lines are returned by each program. These lines are clipped to the unit square (i.e. lines with endpoints outside the square ((0,0),(1,1)) are truncated) and then rendered using Bresenham’s algorithm into a binary image 50 pixels square. The image produced is then compared to the training image of identical size to produce a partial fitness value. The target image is the same size as the template to avoid the increased runtimes incurred by the more usual method of running a small template over a large image. This helps save time and allows the problem to be harder from the point of view of learning the structure of the template, since the template can be much bigger than the ‘scanning’ approach would allow.

The partial fitness value produced is a correlation-based measure and explicitly implements two of the three criteria mentioned in section 3.2. For each corresponding
Table 5.3. Fitness rewards and penalties for template matching

<table>
<thead>
<tr>
<th>Template</th>
<th>Image</th>
<th>ON</th>
<th>OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ON</td>
<td>+1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>OFF</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.1. Training images used for template matching experiment

A parsimony term is employed to control program growth. This is a tunable penalty on the larger individuals within the population. Once the partial fitness value for matching has been calculated for an evolved template, it is scaled by a factor based on the length of that program tree’s genome. This factor due to excessive length is also scaled according to the current best fitness value in the population (the value calculated for matching strength), to allow the overall penalty to scale with the progress of the run and to keep the selection pressure for tree length constant. This fitness value, after
modification, is used for the progress of the genetic search, but is not used when presenting the results. The unmodified fitness values (i.e. with no penalty at all) of the best scoring individuals are used to show the results at the end of the run, demonstrating the ability of the best evolved individuals to solve the problem without reference to their size. Initial experiments without a parsimony control resulted, even with strong probabilistic control over tree generation, in enormous program trees that overflowed the GP software’s limits, despite several increases the in capacity of the system. The parsimony term used was

\[ F_i = U_i - (\max\{U_j, j = 1..M\} \times L_i \times p) \]

where \( M \) is the population size, \( F_i \) is the modified fitness of individual \( i \) after parsimony adjustment, \( U_i \) is the unmodified fitness of that individual, \( p \) is a parsimony scaling term supplied by the user, and \( L_i \) is the length of individual \( i \). Varying \( p \) allows the parsimony pressure to be varied and allows tuning of runs to bias the selection of shorter trees against fitter ones. This parameter can be seen as a way of varying how much improvement a new program tree has to show in terms of fitness before this benefit outweighs any penalty it incurs for increased length.

For these experiments we used two training images in separate instances of the problem. These images were designed to reflect different levels of difficulty for GP by presenting targets with different complexities. The first image is fairly simple and consists of a box with lines across the diagonals, giving a total of six line segments to match. The second image is an iconic representation of a face marking the major features, and has a total of seventeen line segments (see Figure 5.1).

For each experiment, a total of 20 runs were performed each with a population size of 1000. The run parameters can be found in Table 5.2: larger population sizes than those in Chapter 4 were made possible by an improved efficiency in the underlying software system, and an increased mutation rate coming from early observations that convergence of the population was happening too quickly (shown by a flattening of the performance curve well before generation 30) and needed to be delayed. A total of 2 million individuals were evaluated, in a search space of \( 2^{2500} \) (the total number of combinations in a 50 pixel square binary image). The termination criterion stops runs after 100 generations had been completed, regardless of the fitness values obtained.
5.5 Evaluation of results

For each experiment the best individuals from each run were taken and can be seen in Figure 5.4 and Figure 5.5. Plots of the fitness values over time can be found in Figure 5.2 and Figure 5.3, each figure showing both the average performance of the best individuals in all 20 runs, and the performance of the best individual program.

When attempting to model the ‘cross’ image, the best individual scores 162 of a possible 184, attaining this score in generation 97. The average score over 20 runs for generation 100 was 81.15. This is disappointing, being less than half of the attainable maximum, and suggests that even though a very good match was found this is still a hard problem where the chances of success are slim.

For the ‘face’ image, the best individual score was 31 from a possible 132, attaining this score in generation 100. The average performance over all runs in generation 100 was 15.15. These results are very disappointing and demonstrate the difficulty of the image used as a target. To improve the performance of GP on the template matching problem, particularly for the more difficult case where performance is extremely bad, we need to examine the reasons for failure.

One possible reason for failure is the lack of complexity in the evolved trees. A genetic program cannot model an image containing 17 line segments if it cannot grow to contain at least 17 instances of the line primitive (ignoring collinear lines). The introduction of a parsimony constraint might be thought to be a contributing factor in preventing the models from growing to sufficient complexity, but the size of the program trees at the end of the runs was found to be quite large. A look at the best performing program trees however shows that some such restriction may occur. For the cross picture, the average complexity of the trees (in terms of line segments present per program tree) over all 20 runs was 3.3, but many runs exceeded that figure much earlier, going up to 12 line segments in some cases. Only 5 of the 20 runs had best-performing individuals with at least 6 segments by generation 100. For the face picture, average complexity over all runs was 4.6 at generation 100, with peaks of up to 25 lines earlier on. Only a single run managed to produce an individual with more than 17 line segments at generation 100.

The primitive set as used in these experiments also allows many opportunities for improvement to be wasted by permitting useless genetic operations, and allows trees to be generated that have no use. The use of the NullLineList primitive, for example, means that
any trees generated could be quickly terminated by the inclusion of this primitive, reducing the chances of producing a tree that contains a sufficient number of lines to model the image. Additionally, the amount of effort that goes into calculating constant values for the line endpoints or for parameters to the transformation primitives is very large, and can dominate potential sites for genetic operations. Chapter 6 deals with some of these issues as part of a larger discussion on hierarchy in GP, and presents enhanced primitive sets that produce better performance from the same underlying representation.

For both pictures, there is some scope for building blocks to be found in the form of co-ordinate values. Since many of the line segments have co-ordinate points in common, an evolved constant that codes for a correct endpoint could, once found, propagate by being copied to other line segments within the program. This is a single level of structural regularity that could be exploited. At a higher level, whole groups of lines can be replicated and/or transformed to exploit structural regularities in the image such as repeated components.

These results indicate that, especially for high-order instances of this problem, GP is not a particularly effective technique. It would certainly be easier, for instance, to use trial and error on each of the 2500 pixels and this would lead to a constant time solution for any picture. This would only solve this particular instance of the problem, however, and would not produce a symbolic description of the image as GP does. Use of hill-climbing on randomly placed lines might also help rather that the large perturbations caused by the genetic operators. The next chapter outlines how we can improve performance using only existing GP methods by taking a more structured approach to design of the primitive sets, addressing directly many of the problems highlighted here.
a) mean of best performances (20 runs)

b) best individual fitness

Figure 5.2. Fitness profiles for the 'face' picture
a) mean of best performances (20 runs)

b) best individual fitness

Figure 5.3. Fitness profiles for the 'cross' picture
Figure 5.4. Best of run individuals for 'cross' picture

Figure 5.5. Best of run individuals for 'face' picture
Chapter 6

Hierarchy in Genetic Programming

How GP comes up with the solutions it produces to problems is an issue at the heart of its understanding. The powerful representational aspects of GP allow us to produce an almost infinite variety of programs within a relatively constrained structure, but we remain ignorant of how these programs are produced except through genealogical traces of a single program which are rarely enlightening.

This chapter looks at how GP constructs solutions to a particular problem, and how we can use a number of methods to influence this process. The most important factor is choice of primitive set, but this itself encompasses two issues. Firstly, the designer must decide what effect each primitive is to have in terms of transforming its inputs. Secondly, the designer must have some idea of how these primitives are going to work together.

The Building Block Hypothesis [Goldberg, 1989] states that:

"Short, low-order, and highly-fit schemata are sampled, recombined, and resampled to form strings of potentially higher fitness."

In effect, a building block helps solve part of the problem and, when combined with other blocks, more of the problem is solved than either block could do alone. This hypothesis is reminiscent of the classic problem solving technique of divide-and-conquer, where sub-problems are identified and solved before being combined at progressively higher levels to solve larger sub-problems, eventually leading to a solution to the whole problem. This hierarchy of ‘solving power’ is a tempting explanation for how GP
produces solutions, but it is not certain that such a hypothesis is valid for GP [O'Reilly, 1995, Chapter 4].

The important issue of scalability could also be addressed if we knew how, and if, GP built up solutions hierarchically. The ability to produce solutions to large instances of the problem could come directly from extensions of, or modifications to, solutions produced for small instances.

[O'Reilly, 1995, Chapter 3] considers hierarchy in GP. She draws distinctions between various forms of hierarchy. A hierarchical process identifies and promotes useful subtasks, combining them into higher-level components, and producing solutions that are hierarchical in nature. Hierarchical control is a program structure with some kind of observable strategy that divides the tasks at the top level into subtasks, and executes a large task by the efficient sequencing of execution of subtasks. This differs from a hierarchical process in that the former is constructive, producing new solutions from a 'pool' of subtask solutions, whereas the latter is the co-ordination of these partial solutions into a sequence. Hierarchical structure is a static characteristic of the representation of the solutions, in GP the structure is trivially hierarchical because solutions are tree structured. Deeper levels of hierarchical structure can be seen in human-written computer programs that use abstractions in both data and procedure.

A human programmer might use a hierarchical process to produce the complete algorithm for edge detection, as mentioned in Chapter 4 and expanded upon in Chapter 7. From knowledge of the desired output (a binary signal) a necessary top-level task would be the conversion of any output from earlier in the algorithm to a binary one. For this many options are available, one of which would be thresholding. Thresholding would work best if the features that were to exceed the threshold (i.e. come out as positive outputs) were highlighted compared to the rest of the signal. To highlight these features, some kind of signal transformation would be needed that matched the characteristics of the features (edges), thus requiring the development of a suitable filter which would be the supplied way to transform signals. The identification of such subtasks (transformation followed by decision) comes easily to a human from the specification of the outputs by chaining backwards to the specification of the inputs. Hierarchical control comes from the ordering of the elements in the strategy (i.e. transforming a thresholded input signal with a filter is the wrong way round to do it). The implementation, of the top-level ordering and subroutines to perform the subtasks, gives hierarchical structure.
O'Reilly defines a hierarchical solution as one that exhibits both hierarchical control and hierarchical structure. A hierarchical process would be able to construct solutions to a problem with a primitive set that doesn't explicitly encode subtasks as single nodes. If a primitive set does do this, i.e. if it is highly specific, then solutions can be found even without a hierarchical process simply by random combination of components. On a series of experiments evolving sorting programs with primitive sets of varying levels of specificity, it was concluded that although GP can produce hierarchically structured solutions, it does not do so with a hierarchical process. Although GP may stumble across a way to solve the problem that exhibits both hierarchical control and hierarchical structure, it does so through accident and not through a process of subtask identification and promotion.

Even if GP does not operate via a hierarchical process, using hierarchy can still be a good method of increasing the ability of GP to solve a problem. The use of hierarchy cannot be divorced from the notion of abstraction, indeed the very notion of a hierarchy implies the existence of a number of different levels of abstraction organised in a particular fashion, so we can use abstractions available within GP to make use of hierarchy.

The abstraction of procedure within program trees has received a lot of attention due to the promise it holds for scalability (see section 2.5). The idea of producing reusable subroutines for incorporation multiple times within a program tree is that it is only required that the subroutine have a useful effect once, and this effect is then available for free throughout the rest of the tree. Although the mechanics of the various methods (ADFs, Module Acquisition, Adaptive Representation) differ, their aim is the same, to produce solutions with hierarchical character by abstracting out procedures within the solution.

The other kind of abstraction within programs, of representation of data at different levels of abstraction, has not been explicitly dealt with in the GP literature. [O'Reilly, 1995, section 2.3.7] mentions:

"... typing may foster hierarchically controlled solutions by providing additional structure for sub-tasks to evolve and subtask assembly to take place."

but doesn’t investigate this experimentally.
The technique of STGP is capable of producing data types of various levels of abstraction but work on the benefits of STGP so far has concentrated on its ability to improve search by cutting down the search space. This chapter looks at ways of using STGP within the context of a single problem domain, deliberately to *structure* the search space by deliberately reformulating primitive sets with enhanced type systems, such that solutions will have a hierarchy of data abstractions within them. It must be noted that the use of ‘data abstraction’ does *not* mean we will use abstract data types in the sense of encapsulated data accessible only through an interface, nor will we include the ideas of inheritance hierarchies [Haynes, Schoenefeld & Wainright, 1996], although these also may have a place in producing hierarchical solutions. We will use the term ‘hierarchy of representation’ (as distinct from ‘hierarchical representation’ which is easily confused with hierarchical structure) to mean the explicit structuring of program trees such that less complex data structures are to be found at the lower levels of the tree, and these are combined at higher levels through use of more complex data types. This explicit structuring, produced only through careful design of the primitive set and the type system for the problem domain, introduces a form of hierarchy that has not previously been used in GP. We concentrate on how we can use this kind of structuring to improve the performance of GP through both the structuring of the low level data and of the operations performed on it at the higher level.

As a test bed for the experiments, we will continue to use the template matching problem introduced in Chapter 5. This is a good choice of problem because we know every instance is potentially solvable (even trial and error would only take a maximum of 2500 attempts to find any picture by trying each pixel in turn), and we know from Chapter 5 that GP is capable of solving low-order instances of the problem such as the cross picture. We also know that performance on higher-order instances is poor, so we have a good way of judging improvements in performance. All conditions will remain the same i.e. number of runs, population size, training images, but each experiment will use a different primitive set based on an enhanced type system.

It is worth pointing out here that, although from the point of view of the GP software we will be using a number of different types of data, the underlying implementation of the primitives remains virtually identical. In most cases no code changes are needed in the implementation even though we are changing the primitives used. This is because the types introduced into the primitive sets are only used for
structuring purposes, and still map to the same underlying types in the implementation. All
that are changed are the constraints on the process that generates and manipulates the
trees. This is an important point - it shows how strong typing can be used as a general
constraint mechanism of some power without a great deal of extra effort on the part of
the implementer.

Strong typing is not the only way to structure program trees in GP. Use of context
free grammars to define a language used by GP in program trees can also implement such
structuring methods ([Whigham, 1995], [Geyer-Schulz, 1997]). Since we are already
using strong typing to produce and store our return data, however, in the interests of
simplicity we use the same method to perform the structuring.

This work was originally published, in abridged form, as [Harris, 1997a].

6.1 Template matching with hierarchy

A number of enhancements are presented in this section, aimed at improving the quality of
the solutions produced by GP when applied to the problem of matching an image with a
template. These ‘structuring methods’ are mainly a direct response to the problems
identified with earlier approaches in section 5.5. Each of these methods biases the
construction process of the program trees by exploiting the type-safety rules built in to the
GP software, for both initial tree creation and creation of trees through mutation.

6.1.1 Changing the base level representation

The original template matching experiments used a base level representation of a single
line i.e. partial solutions were built up in terms of single lines or unrelated collections of
single lines. Often, interesting images are not unrelated collections of lines but exhibit
some structure, perhaps being collections of shapes, or connected components. Both the
training images used in the experiments exhibit this property. The cross image is a single
connected structure of 6 lines. The face image is 6 connected components, each of
between 2 and 4 lines.

If we use a notion of a connected component as our base level representation, it
might be possible to match more quickly the structure inherent in the image since that
structure would be a built-in property of our primitive set. As a low-level primitive, then,
we can introduce a Component type (the capital ‘C’ refers to the data type used in the
type system, as opposed to the image feature of a connected component which has a small
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Return type</th>
<th>Parameter types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus</td>
<td>double</td>
<td>double, double</td>
<td>arithmetic addition</td>
</tr>
<tr>
<td>minus</td>
<td>double</td>
<td>double, double</td>
<td>arithmetic subtraction</td>
</tr>
<tr>
<td>times</td>
<td>double</td>
<td>double, double</td>
<td>arithmetic multiplication</td>
</tr>
<tr>
<td>divide</td>
<td>double</td>
<td>double, double</td>
<td>protected arithmetic division</td>
</tr>
<tr>
<td>0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 0.33, 0.25</td>
<td>double</td>
<td>none</td>
<td>constants</td>
</tr>
<tr>
<td>And</td>
<td>Component</td>
<td>Component, Component</td>
<td>Joins one Component to another</td>
</tr>
<tr>
<td>Add</td>
<td>VectorList</td>
<td>VectorList, VectorList</td>
<td>Join a number of VectorLists together</td>
</tr>
<tr>
<td>Join</td>
<td>Component</td>
<td>Point, VectorList</td>
<td>Constructs a Component from start point plus list of vectors</td>
</tr>
<tr>
<td>Fragment</td>
<td>VectorList</td>
<td>double, double</td>
<td>Construct a 2-D vector</td>
</tr>
<tr>
<td>StartComponent</td>
<td>Point</td>
<td>double, double</td>
<td>Produce a starting point for a Component</td>
</tr>
<tr>
<td>NullVectors</td>
<td>VectorList</td>
<td>none</td>
<td>A terminator for VectorList groups</td>
</tr>
<tr>
<td>Rotate</td>
<td>Component</td>
<td>Component, double</td>
<td>Rotate a Component anti-clockwise by some amount (in radians)</td>
</tr>
<tr>
<td>XShear</td>
<td>Component</td>
<td>Component, double</td>
<td>Shear a Component in the x direction by some factor</td>
</tr>
<tr>
<td>YShear</td>
<td>Component</td>
<td>Component, double</td>
<td>Shear a Component in the y direction by some factor</td>
</tr>
<tr>
<td>Translate</td>
<td>Component</td>
<td>Component, double</td>
<td>Move a Component around</td>
</tr>
<tr>
<td>Scale</td>
<td>Component</td>
<td>Component, double</td>
<td>Enlarge a Component about its centre of gravity</td>
</tr>
<tr>
<td>NullComponent</td>
<td>Component</td>
<td>none</td>
<td>A terminator used for the And primitive to allow trees to stop growing</td>
</tr>
</tbody>
</table>

'The connected component in the image is formed by drawing a line from the end of the previous segment (or the start point, in the case of the first vector) along the specified vector. For example, a connected component with starting point (0.1,0.1) and vectors ((0.1,0.2), (-0.2, 0.3)) would produce two line segments with co-ordinates ((0.1,0.1), (0.2,0.3)) and ((0.2,0.3), (0.0,0.6)). Each connected component in the image could then feasibly be represented by a single, encapsulate, subtree in the program tree.'
Although in the case of the cross image it is not simple to construct the whole image from a single component (there is no sequence of connected lines that can ‘draw’ the cross in its box without duplicating one or more lines) it is easy to see that it could be represented with only two components. Even components containing only a couple of line segments could be useful as they can be replicated and transformed by the genetic operators, taking advantage of any structural regularity in the image.

The primitive set used for these experiments (called the ‘component set’) is detailed in Table 6.1. This set is extremely similar to the primitives used in Chapter 5, the only difference being the replacement of the line-based primitives with component-based primitives.

6.1.2 Chaining the structure of program trees

The use of the NullComponent, NullLineList (in Chapter 5) and NullVectors primitives in primitive sets is designed to allow branches of the tree to terminate. The type systems require that each of the union operations (And, Add in the component set, Join in the line-based set) take instances of the same type as they return as parameters, to allow an unspecified number of elements to be combined. By making this a binary process, i.e. a join primitive can connect two whole subtrees of base-level elements, we ensure maximum flexibility in terms of combinations, groupings and transformations of those elements.

The binary nature of the join primitives does have a downside, however. At each point where one of these primitives occurs, a Null primitive of the appropriate kind can be used as one of the subtrees. This leads to subtrees full of dead ends, littered with Null primitives which add bloat to the program and contribute nothing to the program’s fitness.

To reduce the incidence of Null primitives and thus reduce the chances of wasting genetic operations, we change the way the join primitives work. Instead of allowing them to combine subtrees of elements in any way, we can use the type system to turn them into ‘handed’ connectives, forming a backbone (see Figure 6.1) of such primitives in a left-handed manner, terminated by a single Null primitive and chaining together base elements on the right-hand subtrees. This increases the chance of having more complex models produced by reducing the incidence of Null primitives in the program tree to single occurrence per chain. The primitive set used for these experiments (called the ‘chaining
6.1.3 Isolating transformations

Once the chaining enhancements were added, it became evident that program trees were dominated by transformation nodes. This is due to the makeup of the subset of the primitives that returns a Component type, which consists of 5 transformations and the Join primitive. This means that tree generation at points requiring a Component is dominated (since selection of primitives is random) by the transformation primitives. To remove this bias, we introduce a new function, Transform, as a conversion primitive.

Each transform primitive is made to return a Transformation type, which is taken by the Transform function and simply returns a Component identical to that produced by the transform function underneath it. This is a type-based sleight of hand, no actual conversion is done on the data, only the type system as seen by GP is altered. This reduces the size of the subset that returns a Component type from 6 to 2, giving the Join primitive a much better chance of being selected.

These new primitives used for the experiments (called the 'transformation constraint' set) are detailed in Table 6.3. This method is only applied to the component representation, since the relevant sections of the type systems are similar for both base-level representations.

6.1.4 Constraining constant generation

A lot of effort goes into calculating constant values to specify line endpoint co-ordinates. This can lead to the proliferation of subtrees dedicated exclusively to constant calculation,
### Primitives for Chaining

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Return type</th>
<th>Parameter types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>ComponentChain</td>
<td>ComponentChain, Component</td>
<td>(as for component set)</td>
</tr>
<tr>
<td>NullComponent</td>
<td>ComponentChain</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>StartComponent</td>
<td>VectorListChain</td>
<td>VectorListChain, VectorList</td>
<td></td>
</tr>
<tr>
<td>NullVectors</td>
<td>VectorListChain</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### a. Primitive set additions/replacements for chaining (component base-level representation)

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Return type</th>
<th>Parameter types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join</td>
<td>LineList</td>
<td>LineList, Line</td>
<td>Join a list of lines (chain)</td>
</tr>
<tr>
<td>Rotate, XShear, YShear, Scale</td>
<td>Line</td>
<td>Line, double</td>
<td>Rotate a Line, shear it in x or y direction, or scale about its centre.</td>
</tr>
<tr>
<td>Translate</td>
<td>Line</td>
<td>Line, double, double</td>
<td>Move a Line around</td>
</tr>
<tr>
<td>line</td>
<td>Line</td>
<td>double, double, double</td>
<td>Construct a Line from endpoints</td>
</tr>
</tbody>
</table>

#### b. Primitive set additions/replacements for chaining (line base-level representation)

Table 6.2. Primitive sets for chaining

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Return type</th>
<th>Parameter types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform</td>
<td>Component</td>
<td>Transformation</td>
<td>Convert a transformed component into type Component</td>
</tr>
<tr>
<td>Rotate</td>
<td>Transformation</td>
<td>Component, double</td>
<td>Rotate a Component by some amount (in radians)</td>
</tr>
<tr>
<td>XShear</td>
<td>Transformation</td>
<td>Component, double</td>
<td>Shear a Component in the x direction by some factor</td>
</tr>
<tr>
<td>YShear</td>
<td>Transformation</td>
<td>Component, double</td>
<td>Shear a Component in the y direction by some factor</td>
</tr>
<tr>
<td>Translate</td>
<td>Transformation</td>
<td>Component, double, double</td>
<td>Move a Component around</td>
</tr>
<tr>
<td>Scale</td>
<td>Transformation</td>
<td>Component, double</td>
<td>Enlarge a Component about its centre</td>
</tr>
</tbody>
</table>

Table 6.3. Primitive set additions/replacements for transformation isolation

which can dominate the genome. This bloat of constant calculations absorbs many genetic operations and reduces the chance of larger scale structural changes occurring through an operation on a higher-level part of the tree.

However, careful use of a type system of numbers can restrict the size of subtrees that calculate the constants. By specifying that the arithmetic functions can only now take
arguments of a new type `ndouble`, and supplying an appropriate set of ndouble terminals (copies of all the terminals of type double), it is possible to restrict the height of any subtree that needs to calculate a double to a maximum of 2. Note it is still possible to have single-node subtrees of type double, as these terminals remain in the primitive set. Again, this is a sleight of hand, since in the underlying implementation these values are all represented as doubles.

With subtrees restricted to such small depths, it is questionable whether GP has the power to calculate all the constant values we might want. It would depend on the number and values of the terminals available. A further level of typing could be introduced by adding another number type `sdouble`, extending the principle of `ndouble` to allow the trees to grow to a maximum of three levels, providing better variability whilst still constraining the size of trees. At this point, the primitive set begins to get rather unwieldy, having large sets of terminals for each number type, although this has no adverse effects on the rest of the type system.

The primitives for this experiment (called the ‘constant calculation constraint’ set) are detailed in Table 6.4. Again, this technique is only applied to the component base-level representations.

### 6.2 Experimental design

Each of these enhancements to the standard method employed in Chapter 5 cannot just be looked at in isolation. Unless the techniques are examined in combination with other techniques, all we can demonstrate is that each is an improvement over the most basic, naïve method. Additionally, all the methods work in separate areas of the type system so should not interfere with each other. For this reason we run a range of experiments each of which uses a unique combination of the enhancements detailed above. This gives a series of 9 new experiments in addition to those in Chapter 5.

1. line-based representation with chaining
2. component-based representation
3. component-based representation with chaining
4. component-based with transformation constraint
5. component-based with constant constraint
6. component-based with chaining and transformation constraint
Table 6.4. Primitive set additions/replacements for constant calculation constraint

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Return type</th>
<th>Parameter types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>plus</td>
<td>double</td>
<td>ndouble, ndouble</td>
<td>arithmetic addition</td>
</tr>
<tr>
<td>minus</td>
<td>double</td>
<td>ndouble, ndouble</td>
<td>arithmetic subtraction</td>
</tr>
<tr>
<td>times</td>
<td>double</td>
<td>ndouble, ndouble</td>
<td>arithmetic multiplication</td>
</tr>
<tr>
<td>divide</td>
<td>double</td>
<td>ndouble, ndouble</td>
<td>protected arithmetic division</td>
</tr>
<tr>
<td>0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 0.33, 0.25</td>
<td>double</td>
<td>none</td>
<td>constants</td>
</tr>
<tr>
<td>0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 0.33, 0.25</td>
<td>ndouble</td>
<td>none</td>
<td>constants</td>
</tr>
</tbody>
</table>

7. component-based with chaining and constant constraint
8. component-based with transformation constraint and constant constraint
9. component-based with chaining and transformation constraint and constant constraint

Set 2 can be compared with the results from Chapter 5 to see how the change of base-level representation affects performance. Sets 1 and 3 can be compared with set 2 and results from Chapter 5 to see how chaining affects performance. Sets 3, 4 and 5 can be compared with set 2 to see how each enhancement on its own affects the performance of GP. Sets 6, 7 and 8 show how the enhancements themselves interact when combined, with set 9 showing how GP fares with all the enhancements added.

For each set of experiments, 20 runs were performed using the face image as a target. As an extra experiment, the component representation was used with the cross image to see if we could improve on the performance already obtained with line-based representations. For our purposes however the cross image really is too simple and can be largely solved with the approach in Chapter 5, so it is not really worth devoting too much effort to find improvements using this picture. The face image is a much more challenging task and therefore appropriate to our needs. We therefore apply the three structuring methods (chaining, transformation isolation and constant sub-tree depth limiting) only to the component representation, and only using the face picture as a target. Other than the primitive sets used, all experiments use the same parameters as in Chapter 5 (Table 5.2).
6.3 Results

The results of the nine experiments described in section 6.2 are summarised in Table 6.5. These show the effects of using the various structuring techniques upon final fitness values at generation 100. Discussion of each technique and its effects follows. As before, the maximum fitness value for the cross picture was 184, and for the face picture, 132.

The best individuals produced for each run, for the component, chaining, transformation isolation, and constant constraint experiments, are shown in Figure 6.7, Figure 6.8, Figure 6.9 and Figure 6.10 respectively (see Chapter 5, Figure 5.5 for comparisons with the line-based, unstructured primitive set).

6.3.1 Changing the base-level representation

Moving from a line-based description to a component-based one seems to have either no effect at all or a minor deleterious effect upon performance. This runs against the intuitive notion that a more appropriate base representation would increase performance. For the cross picture performance differences between the two representations are very small, maximum performance of 167 for the component representation against 162 for the line-based representation, and can be disregarded. Fitness profiles for the line-based and component-based profiles for the two target pictures can be found in Figure 6.2 and Figure 6.3.

For the cross picture, the structural complexity of trees (as measured by the total number of line segments in all the component structures in the individuals) produced by the component set is very similar to that produced by the line-based representation. Average structural complexity produced by the component set at generation 100 was 2.6, with a maximum value of 10. At generation 0, these figures were both 0, meaning that all individuals in the initial generation that produced lines were penalised and gave negative fitness values, whilst individuals producing no lines were given a default fitness value of 0. The figures for generation 100 are sufficient to produce the required model. However, only 2 of the 20 runs produced best individuals with the necessary complexity at generation 100. This compares with average complexity at generation 100 of 3.3, with a maximum of 9, and 5 runs producing sufficiently complex individuals, for the line-based representation.

The 2 individuals produced by the component set that have sufficient complexity to produce the model have 10 and 9 line segments. These line segments are contained
within 8 and 9 component structures respectively. This indicates that the use of components to structure the layout of consecutive segments has little effect for the cross picture. Component structures then become simply a different way of generating single line segments in the majority of cases.

For a simple image such as the cross image, it is equally easy for the GP to generate a new component and add a single line to it, as it is to add an extra line segment to an existing component. The extra structure added by the use of connected components is not needed if a representation based on isolated lines is sufficient to do the job.

When considering the face image, the extra complexity needed in the solutions would be expected to favour the component representation. In terms of complexity, the line-based representation never produces sufficiently complex models to capture significant portions of the training image. At generation 100, average complexity for the line-based representation is 4.6, with a maximum of 22. A minimum of 17 segments is needed to model the image, and only 1 run in 20 manages to produce a best-performing individual with this complexity. For the component set, average complexity was 3.4 with a maximum of 15, so no runs produced a best-performing individual with the necessary complexity. The lack of complexity produced by the trees is addressed in the next section where the structure of the genetic programs is 'chained' to encourage greater numbers of line segments.

The lack of progress associated with the move to a component-based representation illustrates the problem with predicting performance when designing a complex primitive set for use with GP. Although such a representation seems more intuitive, the implementation of the necessary primitives may not be. A single line segment needs four co-ordinate points to evolve to values that produce a net positive fitness for that segment. A component needs two values initially plus a further two for each line segment, meaning that less search effort is needed per line segment (in terms of finding the correct values for constants in the expression) if we are looking at complex models. For single segments both representations need to evolve the same number of constants. In terms of efficiency of representation the component representation looks better because it can represent a complex model with fewer constants. Looking at the program trees shows that this complexity is never used, however. Of the 20 best-of-run individuals at generation 100, only 5 had more than a single component in the tree, and only a single individual had a component containing more than one line segment. This means that the
component structures were not being used to form complex shapes at all, but were simply forming single lines most of the time. The lack of complexity is a major factor in the poor fitness values obtained - the runs with the most complex individuals produced the best fitness values, but the representation could not reliably produce complex individuals.

6.3.2 Chaining the construction of trees

Forcing program trees to be constructed with a ‘backbone’ of join primitives linking a series of data-producing subtrees leads to considerable increases in performance. Fitness differentials for component-based representations can be seen in Figure 6.13, which shows how fitness changes when chaining is added to the plain component-based type system, or to type systems incorporating the transformation isolation and/or constant constraint structuring methods. See Figure 6.4 for the fitness profiles of the component-based system with chaining added.

For the component representation, average best-of-run fitnesses over the 20 runs increase by a factor of three from 9.9 to 30.25, and the best individual overall doubles in performance from 26 to 50. For the line-based representation, average best-of-run performance doubles from 15.5 to 31.2, and best individual overall performance increases from 31 to 45. These results indicate that both methods of producing templates benefit greatly from using chaining, the component representation gaining the most benefit (see Table 6.5, rows 3 and 4). Moreover, performance rises consistently across 20 separate runs, indicating that this method will reliably increase performance so the chances of finding a good solution are better. A method that increases maximum performance (i.e. the greatest performance achieved by a single individual over a number of runs) cannot necessarily be relied upon in practice if average performance doesn’t also increase.

When combined with either of the other two structuring methods used in these experiments, chaining again increases performance (see Table 6.5, also Figure 6.13), with an insignificant decrease observed when adding chaining to both of the other methods in combination. The most dramatic improvement however is the increase in fitness observed when added to the vanilla component-based type system.
Fitness profiles for line-based representation

Fitness profiles for component-based representation

a) line-based representation for the 'cross' picture

b) component-based representation for the 'cross' picture

Figure 6.2. Fitness profiles for line-based and component-based representations on the 'cross' picture
Fitness profiles for line-based representation

![Fitness profile for line-based representation](image)

- Average performance
- Best so far

a) line-based representation for the 'face' picture

Fitness profiles for component-based representation

![Fitness profile for component-based representation](image)

- Average performance
- Best so far

b) component-based representation for the 'face' picture

Figure 6.3. Fitness profiles for line-based and component-based representations on the 'face' picture
Figure 6.4. Fitness profiles for component-based representation with chaining

Figure 6.5. Fitness profiles for component-based representation with transformation isolation
6.3.3 Isolating transformations
Separating transformations from data-producing subtrees via the type system seems to improve performance compared with keeping them of the same type. Compared to the vanilla component set (i.e. with no structuring methods applied) we see a doubling of average best-of-run performance over the 20 runs, from 9.9 to 18.9. The best individual performance jumps from 26 to 34. This structuring method promotes the use of data-producing subtrees by increasing the chances of generating such a subtree during tree construction. Fitness profiles for these experiments can be seen in Figure 6.5.

When combined with chaining, however, performance drops. Average best-of-run performance over 20 runs falls from 30.25 to 26.3. Maximum individual performance drops from 50 to 41.

When transformation isolation is combined with constant constraint performance improves over just isolating the transformations. Average best-of-run performance over 20 runs jumps from 9.6 with constant constraint to 21.9 with both constant constraint and isolation of transforms. Best individual performance increases from 33 to 45. Fitness differentials can be found in Figure 6.14, showing the changes in fitness obtained when adding transformation isolation to the ordinary component-based implementation, and to the other structuring methods used.

6.3.4 Constraining constant generation
The use of strong typing to constrain the size of subtrees that calculate constant values has little effect on fitness when added to the vanilla component set. Average best-of-run fitness over 20 runs decreases from 9.9 to 9.6, with maximum individual performance increasing from 26 to 33. Fitness profiles for the plain component set with constant constraint added can be seen in Figure 6.6.

When added to the chaining method, constant generation constraint causes performances to drop from 30.25 to 28.1 (average over 20 runs) and from 50 to 39 (peak individual). When combined with isolation of transforms, the constraints increase performance slightly, from 18.9 to 21.9 (average, 20 runs) and 34 to 45 (peak). When added to both the chaining and transformation methods, constant generation constraint reduces average performance from 26.3 to 21.55 (see Table 6.5, rows 7 and 10) but increases peak performance from 41 to 44. Fitness differentials can be found in Figure 6.15, showing the changes in fitness when constant constraint is added, both in isolation
Figure 6.6. Fitness profiles for component-based representation with constant constraint

and combined with the other two structuring methods used.

The use of constant constraint appears therefore to be a mixed blessing. When chaining is involved constraining the way constants are generated seems to have a negative effect. However, combining with transformation isolation improves performance. Combining with both has a mixed effect on performance which might be expected from the results with each of the other methods in combination. One explanation for the negative effect on performance when combined with chaining is that the increased complexity of the models resulting from the chaining means that a lot more constants need to be evolved and this could be made more difficult with the additional constraints, leading to less accurate solutions. When added to the transformation constraints, the small trees involved in constant calculation combined with lower complexity in terms of numbers of line segments, would divert more genetic operators to nodes involved in transformations, allowing good solutions to be found by manipulating the transforms of badly-placed line segments.

It is obvious that with a maximum subtree depth of 2, and only a few constants to choose from, it would not be possible for GP to generate all possible values required to model the target image directly with un-transformed line segments. For a fifty pixel square
To see whether the lack of subtree depth was having a serious effect, 20 runs were carried out with an extra level of typing for the constant generation trees (as mentioned in section 6.1.4). This allows subtrees of maximum depth 3 to be produced whilst keeping all other factors constant, which allow a much wider range of calculations to occur within a single subtree. Results from these experiments indicate a slight improvement, average performance moving from 9.9 (vanilla component set) to 10.25 and peak performance increasing from 26 to 34. Such figures however are only a marginal improvement over those gained with depth-2 constraints. Further experimentation to see the effect of combining depth-3 constant constraint with other constraint methods would be necessary to see if this extra flexibility was beneficial across the board.

**Table 6.5. Summarised results of structuring experiment suite (maximum possible score is 132)**

<table>
<thead>
<tr>
<th>Representation</th>
<th>Chaining</th>
<th>Transformation constraint</th>
<th>Constant constraint</th>
<th>Average best of 20 runs at 1 generation 100</th>
<th>Maximum individual performance over 20 runs at generation 100</th>
<th>Fitness change when adding</th>
<th>Fitness change when constraining</th>
<th>Fitness change when constraining</th>
<th>Fitness change when constraining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line Component</td>
<td></td>
<td></td>
<td></td>
<td>15.2</td>
<td>31</td>
<td>+16.1</td>
<td>+10.1</td>
<td>-0.3</td>
<td>+7.0</td>
</tr>
<tr>
<td>Line</td>
<td>✓</td>
<td></td>
<td></td>
<td>9.9</td>
<td>26</td>
<td>+20.4</td>
<td>+24.0</td>
<td>+9.0</td>
<td>+8.0</td>
</tr>
<tr>
<td>Component</td>
<td>✓</td>
<td></td>
<td></td>
<td>31.2</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>30.3</td>
<td>50</td>
<td>-4.0</td>
<td>-9.0</td>
<td>-2.2</td>
<td>-11.0</td>
</tr>
<tr>
<td>Component</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>18.9</td>
<td>34</td>
<td>+7.4</td>
<td>+7.0</td>
<td>+3.0</td>
<td>+11.0</td>
</tr>
<tr>
<td>Component</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>9.6</td>
<td>33</td>
<td>+18.5</td>
<td>+6.0</td>
<td>+12.3</td>
<td>+12.0</td>
</tr>
<tr>
<td>Component</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>26.3</td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>28.1</td>
<td>39</td>
<td>-6.6</td>
<td>+5.0</td>
<td>-4.8</td>
<td>+3.0</td>
</tr>
<tr>
<td>Component</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>21.9</td>
<td>45</td>
<td>-0.3</td>
<td>-1.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>21.6</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

image, fifty different values spread over the range 0.0 to 1.0 would be needed at regular intervals. Adding all these values as primitives, however, would seriously bloat the terminal set and also bias the system in favour of such fifty pixel images. The use of transformations would help to map line segments to positions which could produce positive fitness values.
Another possible way to increase the flexibility of the constant generation without too much extra effort is to add some finer-grained constants to the terminal set. This would allow the quick generation of values such as 0.12 which is hard to produce with a terminal set consisting of relatively large fractions. To test this, 20 additional runs were performed with the extra constants 0.01, 0.02, and 0.05 added. No significant change in performance was noted, however, so the original terminal set was kept.

Using strong typing is a simple way to implement depth constraint on subtrees. Depending on the size of the primitive set being constrained, however, it can be quite inelegant. For depth 2, an additional 13 terminals are required to implement the ndouble type. For depth 3, on top of the ndouble primitives, another 13 terminals and 4 functions are required. This makes for large primitive sets which although easily coped with by the GP system cloud understanding for the designer. Some more explicit regime, perhaps extending the complexity of the tree representation to include depth constraints of subtrees of particular nodes, in the core GP system, might be more appropriate as a practical method. This section does show, however, that strong typing can be very flexible as a method of employing syntactic constraints in GP.
Figure 6.7. Best of run individuals for component set (face picture)

Figure 6.8. Best of run individuals for chaining set (face picture)
Figure 6.9. Best of run individuals for transformation isolation set (face picture)

Figure 6.10. Best of run individuals for constant constraint set (face picture)
6.4 Discussion

These results show that the performance of a set of GP runs can vary widely if methods are used to constrain the architecture of the trees generated. The use of strong typing, a technique already available to the GP designer, is a flexible and powerful way of implementing many types of syntactic constraint of program trees. The experiments performed here show that with a small amount of extra effort, and a minimal amount of re-working of the problem definition and implementation, we can considerably increase program fitness over the levels achieved by the initial representation by constraining the way solutions are constructed using strong typing. More importantly, we have not changed the way that GP solves the problem but only the way in which it constructs the solutions - the underlying implementation and solution space remains the same, but the paths taken through the space are more restricted.

The use of syntactic constraints via strong typing not only cuts down the search space (as any strong typing does), but allows the paths taken within it to be ‘funnelled’ in particular directions. This has been shown to improve performance via three different examples.

6.4.1 The structural complexity of constrained program trees

One important reason for the poor performance in many of the experiments is that the program trees produced simply do not have the necessary complexity, in terms of encoding the requisite number of line segments, to successfully model the target images. The ability of both line-based and component-based representations to model the cross image was due to the relative simplicity of that image in terms of numbers of line-segments required to produce a good model. The face image is nearly three times as complex, requiring 17 line segments instead of just 6 for the cross image.

Methods to increase structural complexity of the models would be useful in at least providing the GP with the chance to find models of the requisite size. If increased structural complexity correlates with increased fitness, then such methods would help us produce better models.

Figure 6.11 shows the structural complexity of the best performing individuals for the plain component representation and for this representation with each of the structuring methods added. Using constant constraint and transformation isolation seems to have little effect on the complexity of individuals, being roughly on a par with the standard
component representation. Chaining, however, significantly increases the complexity of the evolved individuals using identical run parameters. More importantly, chaining is also responsible for the biggest fitness increases of all the structuring methods. These two characteristics of the chaining technique give some empirical support to the notion that increased fitness comes about primarily due to increased structural complexity.

Taking the fitness data of the best performing individuals, for each of the 20 runs, for all 100 generations, and calculating the correlation of these values with the structural complexity of those individuals (for a total of 2000 pairs for each set of experiments), gives the figures shown in Table 6.6. Plots of fitness against complexity, for the best individuals in each generation, can be seen in Figure 6.12. These figures show that fitness and structural complexity do correlate quite strongly, indicating, even for complexity values above the minimum threshold required to model the target image, that increasing complexity can be beneficial.

The notion that increasing structural complexity can help increase fitness is quite obvious in hindsight - the more line segments there are the greater the number of pixels that can be correctly matched, assuming a uniform distribution of line start- and end-points and that unprofitable lines would be selected out. By using the structuring methods we simply increase the chances that any increase in tree size (the more usual definition of the complexity of a genetic program is to do with its size) will also increase structural complexity.

Other methods of increasing tree size do not necessarily have this benefit. A set of experiments was tried with the basic component representation, but with run parameters set so that larger trees would be formed initially and through mutations, but there was no performance increase over the normal set of run parameters despite the increased size of the trees. With identical run parameters but using much larger initial trees, 20 runs produced a peak fitness of 19, with average (best of 20 runs) of 10.65. This result indicates that merely increasing the size of a program tree is not enough, but the method by which the tree size is increased is what matters.

### 6.4.2 Structuring, scalability and building blocks

Constraining tree construction is not a new idea, in particular see [Whigham, 1995]. The use of strong typing to implement constraints is a new approach, however, and provides for a new form of hierarchy in GP. Hierarchical methods improve the ability of programs
Experiment set | Correlation between fitness and structural complexity
--- | ---
component | 0.70
component with chaining | 0.77
component with transformation constraint | 0.82
component with constant constraint | 0.84
component with chaining and transformation constraint | 0.71
component with chaining and constant constraint | 0.69
component with transformation constraint and constant constraint | 0.81
component with chaining and transformation constraint and constant constraint | 0.86

Table 6.6. Correlations between fitness and structural complexity (number of line segments)

to produce solutions to harder instances of problems, that is, they improve the scalability of genetic programs. This technique, which need not be confined to structuring methods such as chaining, isolating transformations, or constraining subtree sizes for constant calculation, is a new way of increasing the scalability properties of GP by promoting abstractions of representation within the tree and as such holds much promise.

In particular, the use of chaining to provide a neutral connective for linking together subtrees provides a rare opportunity to produce genuine building blocks: the data produced by each subtree (i.e. collections of line segments) is simply joined with that of all the other subtrees with a set union operation, rendering the position of the data-producing subtrees irrelevant. With this method it is the content of these subtrees that matter, and not their order. The breaking of epistatic interactions could be an additional explanation for the improved performance when chaining is added, allowing maximum profit to be made from re-use of good subtrees.

It should be noted that using STGP for syntactic constraints is not always the best way of controlling how trees are produced. In particular, isolating transformations is a crude method of specifying the probability of choosing particular primitives at that point in the tree, and this could be specified explicitly as a feature of the underlying software system. Constraining the depth of number-generating subtrees is an important feature and this might also be best achieved with an explicit limitation in the GP software. This shows that STGP is extremely flexible in the type of constraints it can produce.
Figure 6.11. Structural complexity of best evolved individuals with the structuring methods

Figure 6.12. Structural complexity versus fitness for component-based representations
Fitness changes when adding chaining (average)

- Component only
- Component with constant constraint
- Component with transformation isolation
- Component with constant constraint and transformation isolation

a) average best performers over 20 runs

Fitness changes when adding chaining (peak)

- Component only
- Component with constant constraint
- Component with transformation isolation
- Component with constant constraint and transformation isolation

b) peak performance

Figure 6.13. Effect of adding chaining constraint
Fitness changes when adding transformation isolation (average)

![Graph showing average fitness changes with different components and constraints over generations.](image)

- Component only
- Component with constant constraint
- Component with chaining
- Component with chaining and constant constraint

**a) average best performers over 20 runs**

Fitness changes when adding transformation isolation (peak)

![Graph showing peak fitness changes with different components and constraints over generations.](image)

- Component only
- Component with constant constraint
- Component with chaining
- Component with chaining and constant constraint

**b) peak performance**

Figure 6.14. Effect of adding transformation isolation
Fitness changes when adding constant constraint (average)

Fitness changes when adding constant constraint (peak)

a) average best performers over 20 runs

b) peak performance

Figure 6.15. Effect of adding constant constraint
Chapter 7

Complete Edge Detection with GP: Operating Directly on Signals

The two most important pieces of work carried out by the designer when solving a problem with GP are choosing a primitive set and a fitness function. When designing a primitive set it makes sense to use domain knowledge to make the search more likely to succeed. The primitives chosen should reflect the problem domain without making solving the problem trivial, and should allow a rich variety in the way that solutions are constructed so as not to bias the search process and the way in which solutions are found.

For feature detection problems, this means we would expect to gain by using primitives directly related to the signals used as input data. In order to detect features we need to be able to reason about the signals themselves and produce a simple decision or classification from complex data. This means using primitives that refer directly to the signals data and return information about some property of that data. However, in order to develop primitive sets that have some wider applicability than a single problem (see Chapter 3), we want to use primitives that reflect the general-purpose operations used by machine vision researchers today.

In designing the fitness function for these experiments, we continue to adopt the three-component approach, looking for ways of quantifying response strength, response accuracy (localisation) and false positive suppression. We shall return to the problem of edge detection in 1-dimensional signals as a test-bed, as this uses operations such as convolution and thresholding that are widespread in signal processing applications and is
therefore appropriate for our aim of adopting widely applicable primitive sets. This chapter brings together earlier work from this thesis on primitive set design via strong typing, and fitness function design based on properties of generic detectors (see Chapter 3).

7.1 Complete edge detection with GP

Traditional edge detection in digital signals comprises several stages. The first stage is a filtering of the digital signal to highlight the discontinuities by turning them into peaks or troughs by some kind of differentiation (or to look for zero-crossings in some methods where the second derivative of the input signal is taken). This is followed by a non-maximal suppression where only the strongest edge indicators within some neighbourhood are kept and the remainder of the signal is set to zero. In 2-dimensional cases, the non-maximal suppression is followed by a threshold operation to remove edge indicators that are too weak, and finally a hysteresis phase, to join together edge components which look likely to form a single edge in the image.

Chapter 4 concentrated on using GP to produce an optimal convolution filter for the first stage of the process. This work required no domain knowledge as the filters were constructed from the arithmetic operators with a few other numerical functions added. The primitive set used was for a function optimisation problem and, while suitable for evolving a filter, was not relevant to the complete process of edge detection but rather to a single stage in that process. This chapter will tackle the whole edge detection process for 1-dimensional signals as a single problem, taking grey-level signals as the only inputs, and outputting binary signals indicating the position of edge pixels. The work will seek to encapsulate the convolution, thresholding and non-maximal suppression phases of the edge detection process by transforming and analysing the signal in a single operation.

7.1.1 Evolving a complete edge detector

This work seeks to take grey-level input signals and produce binary output signals where 'on' pixels denote the presence of an edge. To do this we need to develop a set of operators that can analyse and manipulate signals, and use a fitness function that assesses the binary outputs we have produced.

The binary nature of the output signals means we must necessarily use a different fitness function from that used in Chapter 4, since terms such as response strength have
little meaning unless integrated over large areas of the signal. We cannot analyse the
output signals produced directly in terms of the three criteria of response strength,
localisation error and noise suppression. We can, however, use these criteria to produce a
fitness function based on correlation with some ‘ground truth’ output signal, giving a
target output which the GP should aim to reproduce. Such an ideal output would be the
result of applying the classic edge detection process to the signals; filtering, performing
non-maximal suppression and thresholding, with a popular choice of filter for the
convolution stage. Following the work on theoretical detectors, we use the filter proposed
by Petrou & Kittler [Petrou & Kittler, 1988] which performs best under theoretical
conditions. This ensures we know that we are at least using a good quality edge detector
system to produce our ground truths, which would also be a likely choice for a machine
vision researcher even though Chapter 4 suggests it may not be optimal in practice. Even
though the evolved operator sets did out-perform the theoretical operators in many cases,
a particular evolved detector for a given ensemble would not be guaranteed to outperform
all other detectors over that ensemble, so we would not be able to say we had an
‘optimal’ detector for all our inputs. Thus we do not aim to use the most specific operator
possible for our ground truths, but to keep the ground truth process constant over all
training sets, and evolve a process for each training set that matches the performance of
the ground truth as closely as possible in the face of different characteristics of the
ensemble used. We are thus trying to emulate what is generally considered to be a very
high-quality edge detection process under a variety of conditions.

This process gives us an ‘edge signal’ which is the associated ground truth for the
grey-level input signal. The fitness function measures how close the output signals
(produced by the genetic programs) are to these edge signals. The aim of this work is to
get GP to do the whole edge detection process in a single step. This means that the
performance of the detector as measured in Chapter 4 is of little significance, since what is
desired is to match an output which is the result of some decision on the part of the
designer as to how edges should be found. Thus the ultimate aim of the work in this
chapter is not necessarily to produce a high-performing edge detector, but to get GP to
emulate the whole process used to produce the ground truth output signals.
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Return type</th>
<th>Parameter types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>threshold</td>
<td>boole</td>
<td>double, double, double</td>
<td>returns True if p1 is between p2 and p3, False otherwise</td>
</tr>
<tr>
<td>mean</td>
<td>double</td>
<td>Signal</td>
<td>mean of the signal</td>
</tr>
<tr>
<td>variance</td>
<td>double</td>
<td>Signal</td>
<td>variance of the signal</td>
</tr>
<tr>
<td>maximum</td>
<td>double</td>
<td>Signal</td>
<td>maximum value in the signal</td>
</tr>
<tr>
<td>minimum</td>
<td>double</td>
<td>Signal</td>
<td>minimum value in the signal</td>
</tr>
<tr>
<td>smooth</td>
<td>filter</td>
<td>double</td>
<td>a Gaussian smoothing filter of specified s.d.</td>
</tr>
<tr>
<td>derivative</td>
<td>filter</td>
<td></td>
<td>a first-derivative box filter</td>
</tr>
<tr>
<td>section</td>
<td>Signal</td>
<td>range</td>
<td>returns a neighbourhood of the current pixel, to size specified</td>
</tr>
<tr>
<td>input</td>
<td>Signal</td>
<td></td>
<td>the entire training signal</td>
</tr>
<tr>
<td>ifn</td>
<td>double</td>
<td>boole, double, double</td>
<td>conditional operator. Returns p2 if p1 True, p3 otherwise</td>
</tr>
<tr>
<td>impulse</td>
<td>double</td>
<td></td>
<td>the value of the current pixel</td>
</tr>
<tr>
<td>convolve</td>
<td>Signal</td>
<td>Signal, filter</td>
<td>convolve a signal with a filter</td>
</tr>
<tr>
<td>3, 5, 7, 9, 11, 13, 15, 17, 19, 21</td>
<td>range</td>
<td>constrained neighbourhood sizes (for use with 'section')</td>
<td></td>
</tr>
<tr>
<td>plus, minus, times, divide</td>
<td>double</td>
<td>double, double</td>
<td>arithmetic operators (division is protected)</td>
</tr>
<tr>
<td>0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 2.0, 0.25, 0.33</td>
<td>double</td>
<td></td>
<td>constants</td>
</tr>
<tr>
<td>lt</td>
<td>boole</td>
<td>double, double</td>
<td>boolean less-than</td>
</tr>
<tr>
<td>gt</td>
<td>boole</td>
<td>double, double</td>
<td>boolean greater-than</td>
</tr>
<tr>
<td>equals</td>
<td>boole</td>
<td>double, double</td>
<td>boolean equality</td>
</tr>
<tr>
<td>or</td>
<td>boole</td>
<td>boole, boole</td>
<td>logical OR</td>
</tr>
<tr>
<td>and</td>
<td>boole</td>
<td>boole, boole</td>
<td>logical AND</td>
</tr>
<tr>
<td>not</td>
<td>boole</td>
<td>boole</td>
<td>logical NOT</td>
</tr>
</tbody>
</table>

Table 7.1. Primitive set for signal-based edge detection

### 7.2 Experimental design

We use several different training sets in these experiments, drawn from those used in Chapter 4.

To produce a binary signal we need a primitive set that can implement a decision process, considering each part of the input signal in turn and returning a yes/no answer as to whether there is an edge at that location. This suggests that we need some way of turning numerical data produced by operations on signal data into binary data. An obvious way to do this is to introduce a boolean type and a method to map numbers to boolean...
values. The most useful way to do this is to use a thresholding function that returns boolean True when the input falls into a certain range and False otherwise. The introduction of boolean values means we need ways of manipulating these values, so some boolean connectives are added.

To manipulate the signal data, we take inspiration directly from the way such information is gleaned in vision research. Signals are transformed by convolution filters, and statistics can be extracted to quantify various properties of these signals. Adding a convolution primitive allows us to transform any signal in a variety of ways. We also use simple statistical measures such as mean and variance to extract information from the signals. Since edges are localised features (albeit over a range of scales), we know we want to be looking at local properties of the signal, so must also provide ways to access the signal data directly and to extract neighbourhoods surrounding the current pixel. To this end we add an impulse primitive which returns the signal value at the current point, and a neighbourhood operator which returns short signals of various sizes around the current pixel. The numbers gained from these operations can be manipulated with the arithmetic operators, as well as some comparison operators (less than, greater than) and a conditional which should allow some reasoning. The full primitive set can be found in Table 7.1.

Each signal in the training set is processed by every candidate solution in the population. The fitness scores produced are added together to produce a single raw fitness score, a higher score indicating better performance. For each signal, the program tree is executed once for each pixel and returns a boolean value. These boolean values are collected together to produce an output signal. This signal is compared with the ground truth edge signal, the output of applying a conventional edge detector to the input signal (see section 7.1), to produce a fitness value.

The fitness function uses the three component design outlined in Chapter 3, having elements to measure response strength, response accuracy, and noise suppression. The fitness value is formed in the following way.

1. The number of True responses in the program’s output signal $S_P$ is totalled, measuring the strength of response from the program, $R_P$.

2. For each edge position in the ground truth signal $S_G$, the offset of the nearest edge found in $S_P$ is calculated. These are rated for accuracy - an edge in the
Objective: To produce a genetic program that produces binary output signals matching the outputs of a target edge detector when applied to grey-level 1-D signals.

Terminal Set: see Table 7.1
Function Set: see Table 7.1
Return type: boolean
Fitness Cases: 3 training sets of 4 signals each from different image ensembles, each totalling between 1100 and 1200 pixels

| Raw Fitness: | a 3-component function detailed in section 7.2 |
| Standard Fitness: | (Max raw fitness) - raw fitness |
| Population size: | 500 |
| No. generations: | 50 |
| Selection method: | Tournament, size 2 |
| Mutation rate: | 20% |
| Crossover rate: | 70% |
| Parsimony factor: | 0.01 |
| Selection method: | tournament, size 2 |

Table 7.2. Tableau for the complete edge detection problem

right location is given a score of 5, a 1-pixel error gives a score of 3, a 2-pixel error gives a score of 1. Larger errors are not rewarded. These scores are totalled to give a localisation score $L_p$.

3. The negative of the absolute difference between the number of edges in the ground truth signal $R_G$ and the number of edges in the response $R_p$ is calculated, giving a score $N_p$. This ensures that signals are penalised if they return too few or too many edges, incorporating both response strength and noise suppression measures by rewarding an accurate number of edge responses.

4. Overall fitness is calculated as $N_p + L_p$.

This function rewards the accurate location of edges and encourages signals to return the correct number of edges as well. Because of the difficulties with measuring response strength in a binary signal, this criterion is combined with the noise suppression criterion to produce an overall measure indicating the disparity in positive response counts between the evolved detector's output and the ground truth output signals. A small parsimony factor was added to encourage shorter solutions to develop in an identical manner to that used in Chapter 5 (see section 5.4). Fitness values for individuals depend
on the number of edges in the signals, and therefore are not comparable between training
sets, but are perfectly valid for the competitive environment used in GP.

Three sets of experiments were performed. Each set used identical conditions and
run parameters but used a different training set of signals. The training sets are the same
sets as used in Chapter 4, consisting of synthetic step edges corrupted with noise, profiles
taken from underwater surveillance pictures, and profiles taken from pictures of an office
environment. These sets will be called the synthetic, underwater, and office sets
respectively. Since the computational requirements for these experiments are
considerable, the size of the training sets was reduced to allow runs to complete in a
reasonable amount of time. Each training set consisted of four signals selected from the
same sets of signals as used in Chapter 4. Four signals is a small number but this equates
to many hundreds of points along those signals which need to be correctly classified, so
the actual number of fitness cases is extremely large. For each set of experiments, 10 runs
were performed. Additionally, a single preliminary run was performed as a test of
concept, using eight step edge signals with no corruption. A tableau for these experiments
can be found in Table 7.2.

7.3 Computational efficiency of signal-based primitives

The use of primitives that calculate information based on signal data has implications for
computing run-times. In particular, primitives that produce statistics on signals can be
quite time consuming when the signals themselves are large. The computation of mean,
variance etc. can have a large effect on run-times, particularly if done often and on large
signals. This is exacerbated by the size of the populations we use and the number of
generations and training signals in the training set.

It is important therefore to ensure that code is efficient and as little duplication of
effort is made as possible. The choice of a low-level language such as C++ to implement
the primitives is useful, as it allows compilation of the intensive code to produce fast
executables. More important still are efforts to make the execution of genetic programs
efficient by coding the underlying structures in an efficient manner. There are certain
lower bounds of effort which cannot be improved upon, for instance when calculating a
statistic of a signal it is necessary to consider every pixel in that signal, but improvements
in performance can be made by small-scale caching of results and only doing calculations
when necessary. To this end, the underlying data structures representing signals keep a
running record of the statistics used in the primitive set, and recalculate them only when
the signal is changed in some way i.e. after convolution. This saves time when signals are
copied and passed around since the statistics can be copied directly rather than being
recalculated.

The primitive set is such that we cannot really save much time on top of this in
terms of efficiency. Since we can make no predictions about how or whether signals are
going to be used by nodes further up the tree, and indeed which signals are going to be
used (since the primitive set allows for taking small areas of existing signals), we cannot
identify areas where we can delay executions or calculations. This is one area where a
functional language with lazy evaluation could have advantages as an underlying
implementation, although such advantage may be outweighed by other run-time
inefficiencies of such a system. Profiling of the system indicates that about 40% of run-
time is spent in copying signals to and from program stacks, which is unavoidable in the
implementation used and would be extremely hard to cut down given the ability of the
primitive set to generate new signals dynamically.

7.4 Results

The quality of the best detectors produced by GP is best illustrated by the ideal outputs
and the outputs produced by those detectors. Each set of results presented consists of a
'strip', which shows the output of a particular detector on an input signal. The 'strips'
have been widened for better visibility, with a single column of pixels in each strip
representing a single pixel of the output signal. Black pixels represent negative responses
i.e. no edge, white pixels represent positive edge responses. For the test run all signals are
shown for the best individual produced. A perfect result by an evolved detector would
have all the white stripes present in the 'ideal' detector, and no spurious stripes.

For the three full sets of experiments, results are shown for the best individuals
produced by the best 3 of the 10 runs performed to show the types of performance
produced. Results are also shown for each set of evolved operators, using a test set of
signals drawn from the same ensemble as the training signals, which were not part of the
training set used to produce the evolved detectors. The quality of these results shows the
generalisation ability of the evolved detectors.
Figure 7.1. Edge detection results for synthetic step edge training signals, no noise

7.4.1 Preliminary ‘test’ run

This run used signals containing only step edges, with no noise corruption whatsoever. With only a single run a perfect performer was found, at generation 9 of 50. The individual correctly classified every edge with localisation error of not more than one pixel, and produced no false responses. Strips for the ideal and output signals can be seen in Figure 7.1. The genome of the best individual was

\[ \text{gt(mean(section(3)),impulse)} \]

which can be read as “return True if the current pixel is greater than the average of the three pixel neighbourhood of the current position, False otherwise”. This is a simple way of highlighting discontinuities in the input signal, and will always produce a True value on the higher-valued of the two pixels that border an edge.

7.4.2 Synthetic signals corrupted with Gaussian noise

Results for these detectors are presented in strip form in Figure 7.2 and Figure 7.3 for the training set and test sets respectively. On the whole the evolved operators seem to match the ideal outputs very closely indeed, operator 1 (the operator that produced evolved outputs (1) in the figures) being the most likely to match the ideal outputs. Operator 2 performed well but missed edges on two of the four training signals, and three of the four test signals. Operator 3 missed only a single edge that was present, on signal 4 of the test set, but was most prone to producing spurious edges not located by the ideal detector. These spurious edges for operator 3 are to be found on signal 3 of both the training and
Figure 7.2. Edge detection results for synthetic edges training set

Figure 7.3. Edge detection results for synthetic edges test set

test set, and for signal 4 of the training set. Overall fitness values for the three operators were 201, 194 and 203 respectively, from a possible maximum of 265. The genomes for the individuals were:

Operator 1:
\[
gt\{\text{absolute(absolute(minus(minimum(\text{convolve(input,derivative)}))}, \text{mean(\text{convolve(input,smooth(plus(variance(\text{convolve(input,smooth(plus(variance(\text{section(3)})) times(minus(minimum(\text{convolve(input,smooth(variance(\text{section(17)}))}, \text{ifn(gt(variance(\text{section(3)}),impulse),0.7,0.4)),0.1)))),times(minus(minimum(\text{convolve(\text{convolve(\text{convolve(\text{convolve(input,derivative)}}, \text{derivative),smooth(0.3)}},smooth(impulse))))),variance(\text{section(3)}))})})})})})}
\]

Operator 2:
\[
\text{threshold(0.1,minus(maximum(\text{section(3)}),minimum(input)),ifn(not(gt(divide(plus(0.25,times(0.2,variance(\text{convolve(\text{section(13)}}, \text{smooth(divide(0.25,variance(\text{section(3)})))))),\text{impulse}),divide(\text{minimum(\text{section(9)}),absolute(variance(\text{section(3)})))),\text{divide(\text{maximum(\text{convolve(input,derivative)}),0.25}),minus(\text{impulse,minus(\text{maximum(\text{convolve(input,derivative)}),1.0)}))})}))})})}
\]
7.4.3 Office environment signals

These signals differ from the synthetic signals in that they contain higher levels of noise and less distinct edges.

Around areas where edges were expected good responses from the evolved detectors were found, however the incidence of spurious positive edge markings was greatly increased compared to that of the operators evolved on the synthetic images. As noise suppression normally involves some kind of smoothing i.e. looking at the signal after some of the high-frequency data has been removed, reduced localisation performance is also expected. Many of the output signals show wide edge responses of several pixels, indicating that several pixels around the true edge position were thought to be suitable edge candidates. This is particularly the case where many edges are located closely together and can interfere with each other, most markedly in signal 2 of the test set.

Although it is usually extremely hard to tell what an evolved detector is doing simply from the genome of that detector, it can be seen from the output signals that the best evolved detectors tended to be more sensitive than the ideal detector. This means they were less likely to miss a genuine edge by virtue of being more likely to produce a positive output. The downside of this is that the evolved detectors had a tendency to produce false positives and to mark edges with thick bands of positive pixels rather than just a single positive output. This reflects the general problem of where to mark an edge - usually the edge pixels to either side of the discontinuity in the input are equally valid as places to mark the edge, so a simple decision process might decide that both of them are to be marked. Fitness values for the detectors were 245, 248 and 276 respectively, from a possible maximum of 415.
The genomes for the three detectors were:

**Operator 1:**
\[
\text{lt} \left( \text{lt} \left( \text{gt} \left( \text{minimum}(\text{section}(19)), 0.5 \right), \text{variance}(\text{section}(3)), 0.3 \right), \text{divide} \left( \text{divide} \left( \text{minus} \left( \text{minimum}(\text{section}(9)), \text{maximum}(\text{section}(19)) \right), 0.1 \right), 0.3 \right) \right)
\]

**Operator 2:**
\[
\text{and} \left( \text{threshold}(\text{variance}(\text{section}(11))), \text{maximum}(\text{convolve}(\text{section}(11), \text{smooth}(\text{absolute}(\text{maximum}(\text{convolve}(\text{section}(11), \text{smooth}(\text{impulse}))))))), \text{impulse} \right), \text{threshold}(\text{variance}(\text{section}(3)), 1.0, \text{maximum}(\text{convolve}(\text{section}(11), \text{smooth}(\text{minimum}(\text{convolve}(\text{input}, \text{smooth}(1.0))))))) \right)
\]

**Operator 3:**
\[
\text{threshold}(\text{variance}(\text{section}(3)), 0.9, \text{ifn}(\text{lt}(\text{maximum}(\text{section}(5)), \text{plus}(\text{maximum}(\text{section}(3)), \text{times}(0.1, \text{impulse}))), \text{ifn}(\text{threshold}(\text{impulse}, \text{impulse}, \text{maximum}(\text{section}(5))))) \text{minimum}(\text{section}(5)), \text{divide}(0.7, 0.1)), \text{divide}(\text{maximum}(\text{section}(5)), \text{maximum}(\text{section}(3))))
\]

### 7.4.4 Underwater environment signals

Due to the noisy and blurred nature of these signals, few edges were present in either the training set or test set. Many of the edges produced, again, consisted of wider areas of positive response than those produced by the ideal detector, reflecting the need on the
part of the evolved operators to consider bigger areas of signal in order to make a decision. This often led to the evolved detectors failing to pick up some of the weaker or narrower edges, whilst over-responding to others.

Encouraging results can be seen for signal 3 of the training set. This signal was so indistinct that no edges were marked at all by the ideal detector. This result was matched perfectly by all three of the evolved detectors, reflecting the low level of false positives produced over all the signals used. Any false positives that did occur were usually in the neighbourhood of existing positive results where conditions were still quite similar, reflecting the poor localisation qualities of these detectors. One notable exception to this can be seen for operators 2 and 3, for signal 1 of the training set. This low false positive rate is in contrast to the office detectors, which produced many false positives. Fitness value for the detectors were 70, 63, and 67 respectively, from a possible maximum of 130.

The genomes for the evolved detectors were:
7.5 Discussion

The process of edge detection is a complex one and certainly extremely hard for GP to emulate in its entirety. The results presented in this chapter, while encouraging, indicate that this problem might be too hard for GP to tackle.

The algorithm used by the ground truth detector consists of a transformation of the input (the convolution), followed by a post-processing stage (non-maximal suppression), and finally a decision process (thresholding). The primitive set used is capable of doing the first and third stages in a straightforward manner, indeed this was one of the motivations behind the choice of primitive set. The second stage is more difficult and to do this directly would require a more complex approach to the problem. Non-maximal suppression for a particular pixel requires access to an intermediate output signal which would contain the results of convolution of the input signal with some kernel. The program trees are not given access to this data (they can work on local data and global input data only, and do not have access to the outputs from the processing of other pixels in the input signal) so cannot perform the equivalent operation to non-maximal suppression in the way the ground truth detector does. Expecting GP to perform a non-local procedure on the basis of just local information is therefore optimistic.
A more successful approach which allowed access to non-local information and/or intermediate representations would need be able to work on entire signals, rather than single pixels. Additionally, access to the previous decisions of the operator (elements of the output signal that had already been decided) would allow coherence properties in the outputs to be evolved and exploited. A wider range of filter types could be provided, in addition to differentiation and Gaussian smoothing, to permit other transformations of the input signal. Primitives to iterate over signals, using evolved algorithms to transform those signals, would allow such global transformations to be evolved. Combined with access primitives for these signals, such abilities would provide for extremely flexible operators to be evolved.

That the evolved detectors performed as well as they did is encouraging. It shows that a correlation-based fitness function can optimise genetic programs performing complex procedures. Although such detectors are easy to evolve for perfect inputs, such as the test run, things get more complicated for the harder training sets. Here, the detectors have to be tuned to cope with varying (although it is assumed they are bounded) levels of noise and to cope with uncertainties over edge positions. Since the primitive set is incapable of emulating the algorithm of the ideal detector by using a similar approach (it cannot directly implement non-maximal suppression), it is to GP’s credit that it managed to simulate it as well as it did.
Chapter 8

Conclusions

GP’s credibility relies on its ability to solve difficult, real-world problems with the minimal amount of human effort. If it cannot make the successful transition from academic exercise to industrial application it will be abandoned in favour of other machine learning techniques.

GP has a lot of promise due to its representational flexibility. If a computer is to be used to address a problem, GP in theory could produce the software that the computer runs. Chapter 4 shows that even in its most basic form GP can produce high-quality solutions to difficult problems. GP is a relatively young technique however and doesn’t yet have the power that we would like. While we know that genetic search is a very good method of exploring large spaces, GP still appears ineffective on some problems because it doesn’t use the ‘search energy’ available to it in the best ways. In order to produce successful results on hard problems, researchers are often forced to make the problems easier, either by scaling down the order of the problem or by over-specifying possible solutions in the form of powerful, highly problem-specific, primitive sets.

For many problems this cannot be done and is not desirable. Often, low-order instances of problems are easily solvable by other means and GP should not be used to tackle them. The work in Chapter 5 shows a problem which GP, with all its representational power, is not suited to solve despite it being very easy to solve using a simple exhaustive search. It is possible to over-specify a solution to a problem only when a solution form is known in advance, and there exist many interesting problems where such a form is not known (optimisation problems being an obvious example - see Chapter
This thesis has shown that GP is capable of producing very high quality solutions to difficult problems. Work in Chapter 4 has produced edge detection filters that outperform those designed by human experts in the digital domain. Further, the method used can produce detectors tailored to the properties of the data upon which they were trained, allowing the production of bespoke detectors.

The use of more generic primitive sets, applicable to a whole class of problems, benefits GP by reducing its dependence on highly-specific domain knowledge and human implementations of custom primitives for single problem domains. Chapter 7 presents initial work on a generalised primitive set for signal processing tasks.

Such generic primitive sets would require more search power on the part of GP to be able to function properly, however (see [O'Reilly, 1995, Chapter 3]). Thus the development of generic primitive sets was accompanied by the development of a new technique for increasing the scalability of GP through abstraction of data within the program tree. The importance of abstraction within GP lies in its ability to get GP to solve high-order instances of problems by capturing properties of the solution and re-using them correctly. [Langdon, 1996a, p225] referring to his work on building structured memory in GP, states:

"While GP work to date has concentrated on functional abstraction, we argue that techniques which allow GP to take advantage of data abstraction will be essential to enable it to scale up and tackle large, real problems."

Work in this thesis in Chapter 6 tackles the issues of data abstraction from a higher-level point of view, dealing with the construction of program trees. Such techniques are a new form of abstraction in GP and are a further aid to the GP’s scaling abilities. Abstraction techniques need to be widely adopted if GP is to scale properly and to become a useful real-world tool. This abstraction is an application of syntactic constraints implemented through strong typing, which are a powerful way to improve the performance of GP by directing the paths taken through the solution space, making the search smarter. This application of STGP is not restricted to abstraction but can be used in many ways, as demonstrated in Chapter 6.

Vision problems are a class of problems where solution forms are often not known.
in advance and could benefit greatly from a powerful exploratory technique such as GP. Such problems make the choice of fitness function extremely difficult if the ideal solution form is not known. By concentrating on the properties of the ideal detector, rather than its form, Chapter 3 presented a framework for designing fitness functions for detection problems which was subsequently used to design three fitness functions in practical work.

Even if GP doesn’t produce perfect solutions, it can produce ones which are within some specified tolerances, and the symbolic nature of the results permits some kind of analysis (Chapter 7 uses primitive sets aimed at adopting the techniques used by computer vision engineers). GP can therefore be a useful tool for suggesting engineering approaches to solving real-world problems by generating and testing potential solution strategies. Vision problems represent an ideal test-bed for pushing the limits of GP.

8.1 The future of GP

Canonical GP, as must be expected of a young technique, needs improvement if it is to fulfil its potential. The wide range of techniques described in Chapter 2 help to extend the power and applicability of GP. Further enhancements could be made:

- Make the genetic search process self-monitoring and adaptive. GP runs are often considered in isolation, as discrete ‘chunks of search’, and are abandoned if they fail to achieve an objective. Improvements in overall performance could be made if the results of each run were used to seed populations for further search, with some cross-pollination from different starting points in the solution space, producing a library of possible solution strategies which can be exploited by later runs. This approach can be seen to some extent in genetic search with demes and should be encouraged. The tendency of populations to converge and stagnate can be caught if some measure of population uniformity is used - various measures have been proposed based on phenotypic or genotypic similarity. Once a population is seen to be converging, it can be disrupted in many ways to introduce some variety whilst saving the best individuals for future reference and exploitation. Techniques such as Limited Error Fitness [Gathercole & Ross, 1997] can help keep a search from converging by adaptively changing the environmental pressures on the individuals concerned.

- Good software support can help the experimenter greatly. In particular,
transparent support for large-scale distributed processing can allow the GP researcher to tackle higher-order problem instances. Many problems have to be 'cut back' due to a lack of resources and this makes them less interesting.

- With new levels of computational resources and better search techniques, the development of generalised primitive sets can take place. These would be applicable to a wider class of problems and would not contain any over-specified, highly problem-specific functions. Applying such primitive sets to a range of problems would allow the development of measurements of problem hardness between different problems, since the representation and search process remain constant. With generic forms of fitness function added (such as correlation-based methods) inter-problem comparisons would be even more valid. Extra resources could be utilised by the development of massively parallel execution environments using the World Wide Web [Hulse, Gerber & Price, 1997].

8.2 Future work

There are many ways in which the work in this thesis can be developed. Primarily the techniques developed should be extended and applied to a wider range of problems within machine vision. To do this, the primitive set used in Chapter 7 would need to be refined and extended. Adding non-local primitives which can operate on arbitrary parts of input signals (as opposed to either the whole signal or just a local neighbourhood) would free the experimenter to process entire signals with a single call to the program tree, rather than work at the pixel level. This, if combined with more flexible representations within the program trees such as looping constructs and some method for storing state, would allow a great deal of flexibility in approaches to algorithm evolution. Most immediately the work should be extended to treat 2-dimensional data directly.

The template matching problem is inherently unsuited to GP in the form used in this thesis. As a platform for demonstrating the structuring methods it is quite suitable, but to tackle the problem seriously with GP would require drastic reformulation of the problem, using both a different primitive set and fitness function. Since recovering shape information is an important problem, fruitful work could be done in this area.

The application of structuring methods via strong typing, however, should be
developed. The chaining method is applicable to other problem domains where some kind of structured tree growth is desirable. The ability to use this method, along with container classes, to generate arbitrary amounts of data could be a good approach to many other problems. One important outcome of this method is that the pieces of data produced by program trees with chaining are building blocks, in the true sense of the term as used in the Building Block Hypothesis. The use of essentially ‘neutral’ connective functions, which do not alter in any way the results produced by their subtrees (as is the case with chaining, but not without it), gives each data-returning subtree the position-independence required to sever epistatic links in the program tree. Use of this technique in other problem domains therefore may lead to improved performance as the necessary conditions for the BBH to apply are fulfilled.

The transformation isolation method used in Chapter 6 is an attempt to influence the selection frequency of a particular primitive, crudely implemented with strong typing. Although this does show how flexible strong typing can be as a constraint mechanism, interesting work could be done exploring the idea of adding selection frequency mechanisms to the GP kernel, allowing the user to state explicitly how useful the primitive is expected to be. Further work could make this adaptive based on correlations between fitness values and the frequencies of each primitive.

The third structuring technique, constraining the size of subtrees that calculate constants, illustrates a very real problem in GP. Many problems require solutions involving some kind of numerical parameters, which need to be evolved. Finding the right values for such parameters is not the sort of job that GP is very good at, so investigating the use of hybrid techniques (combining GP with another search technique i.e. simulated annealing) could be fruitful for this problem. Work in this area has already been done for other problem domains ([O'Reilly, 1995, Chapter 6], [Esparcia-Alcazar & Sharman, 1997a]).

Finally, some work on the approach to tree generation outlined in Appendix A should be done to see if it offers an advantage over standard methods in terms of performance. Although it is useful as a way of reducing the number of run parameters that need to be fixed, a study of performance would be valuable.
Bibliography


160


Appendix A: A practical tool for Genetic Programming

A.1 Introduction

GP software systems play an important role in determining the usefulness of GP as a method of automatic programming. A proper software support structure is essential if GP is ever to become a viable alternative to manual methods in problem solving. This chapter is intended to highlight issues of software construction and portability in GP systems in heterogeneous computing environments, and to propose a framework in which new techniques and methods might be incorporated into a system with minimum disruption. Such a framework might facilitate widespread uptake of these new techniques.

GP conceptually operates at two levels. The technique itself is represented by the cycle as shown in Figure A.1, consisting of a repeating sequence of discrete stages. Some of these stages are basically independent of the problem domain, and perform the necessary manipulations of populations of program trees according to either random values at the start of a run, or fitness values as the run progresses. This part of the cycle does not change, even from problem to problem. Other stages, which perform the mapping from genome to fitness value, are dependent on the problem to which GP is being applied. This part of the cycle often gets completely replaced as new problem domains are addressed.
A.2 Structure of a GP system

Genetic Programming is a cyclical process, based on the evaluation and modification of populations of potential solutions. Each component of this process implements a specific function in the operation of the system as a whole. In a given implementation several components may be combined into a single unit, and in extreme cases the whole system may be integrated, but the genetic programming software system always contains elements that operate at three levels.

- Domain-independent software, which remains unchanged regardless of the problem domain to which GP is being applied. In this category fall the standard repertoire of genetic operators, selection methods, and population storage and manipulation code.

- Problem-specific software changes according to the problem domain to which GP is being applied. This would include problem-specific caddies for wrapping up the evolved individuals in some context, fitness functions, code for reading and selecting fitness cases. For instance, the work in Chapter 4 evolves a single function which when executed produces a number based on some global variable $x$. This needs to be executed several times whilst varying $x$ to produce an array of data used to encode the filter, which then needs to be normalised and convolved with a signal. The actual evolved code is therefore very small compared to the rest of the problem 'caddy'.

- Run-specific software is different for each experimental run of a specific problem. This would be fairly small compared to the other kinds of software present as most runs are performed under identical conditions, but would mainly be involved with varying run parameters such as selection method, operator frequencies, and possibly tuning parameters to the fitness function. This software would be varied to find a good set of run parameters which would then be used for further experimentation.
Figure A.1. Conceptual structure of the GP cycle

This distinction is a hierarchical one and maps neatly to the conceptual structure of the GP cycle: domain-independent software is used for all problem domains, each of which can have any number of independent experimental runs. GP software is unusual in that it is applied to so many different problem areas and is itself highly configurable. It demands the inter-operation of stable (domain-independent, often supplied as standard by the system author) and volatile (domain-dependent, written by the practitioner) software units as the experimenter moves from one problem domain to another. The structure of the system should be such as to allow these units to co-operate with the minimum of effort.

A.2.1 Problem-dependent components

To apply GP to a particular problem, a number of components need to be defined that are unique to the problem domain. These components constitute the definition of that particular problem, and determine the performance of the GP system with respect to that problem in terms of both computation time required and of the fitness of evolved programs.

- Problem domain definition and implementation. This is software that models the problem domain and produces the results of applying the candidate programs to the problem data. This component requires a large
amount of effort in terms of setting up an appropriate problem representation.

- Raw fitness calculation. This produces some scalar value indicating the performance of each candidate on the given problem by analysing the results produced when the genetic programs are evaluated over the fitness cases.

- Configuration. Problem-specific and run-specific settings for the system as a whole. This includes things like the relative frequency of genetic operators to be applied to the population, selection scheme to be used, population size etc.

A.2.2 Problem-independent components

These components are pieces of software that do not need to be changed, regardless of to which problem the system is being applied, or which run is active. These components are the global software mentioned in the introduction.

- A population manager stores candidate programs and renders them in a form suitable for execution. It also performs the necessary operations to produce a new population from the old population. Finally this component provides permanent storage facilities, and retrieval, for recording populations.

- Fitness calculation. The methods for calculating the fitness of an individual for selection purposes are standard and rely only on the raw fitness values produced by the problem-level software.

- Termination criterion. A test to decide whether a particular run has finished or not. This can be a generation count, or something based on fitness values. This component can often be drawn from a stock of standard methods, or (less frequently) may be a custom property of the problem domain in which case it would fall under problem-dependent software.

A.3 The GP-COM software system

GP-COM is a component-based system the structure of which matches closely the conceptual divide between the domain-dependent and domain-independent components. It was designed with the intention of minimising the amount of effort required to encode a new problem and solve it using GP, whilst minimising run-times. GP-COM consists of
a suite of programs glued together by higher-level scripts. Each component of the generic
system described earlier is represented by one or more independent programs. These
programs are controlled by scripts which handle inter-process communication and
sequencing of events to implement the GP cycle. Creating a new problem involves
creating new versions of those elements that are problem specific. The top-level structure
of GP-COM mirrors the conceptual structure of a GP software system and can be seen in
Figure A.2.

A.3.1 System Structure

GP-COM is a loose collection of programs, each with tightly specified input/output
interfaces (see Figure A.4). With GP-COM the only programming effort required on the
part of the GP practitioner is the application of the candidate programs to produce raw
fitness values. Everything else in the system is problem-independent and automatically
applied. The hierarchical, inherited nature of the system saves time and effort by
maximising the re-use of common software components. Three levels of control are
implemented, using the TCL scripting language and TK interface toolkit, and various
C++ programs.

The most abstract level of control is at the system level. This is a simple script
allowing the user to specify the problem on which to work. Problem-specific software is
left to the user to implement. Global properties and configuration is available to be
copied to specific problems.

The next layer of control is at the problem level. The manager process is a script
that launches an instance of the population manager. This script implements job
management (farming out to other machines), and software to control the creation,
execution, and configuration of a number of runs of the same problem. Creation of a new
run involves generating appropriate random seeds, and creating the file and software
structure which the run will use. The code base used by each run is inherited from
problem-specific software, as each run can be viewed as an instantiation of the problem.
To start a run, the run manager first searches for an available host and then launches the
run on that host via a remote shell using the UNIX ‘rsh’ command.

Run-level control is encapsulated inside the manager process. This TCL script
implements the whole GP cycle for a given run. It supports multiple architectures,
running system-specific versions of the problem-specific executables. The problem
program is always architecture specific, so a copy of the program is made for each architecture and naming conventions allow the manager process to choose the correct executable for whichever machines are to be used as workhorses. The main part of this script involves communication with the population manager, which is performed using a pipe and a simple command protocol to effect changes in the stored populations. The population manager is kept running in the background while the other parts of the cycle are executed in turn. Execution of the problem program, fitness calculation and processing, population and fitness logging are performed, and the termination criterion is tested. Selection and genetic operation commands are generated and piped to the population program. The new population is written to disk, and the cycle starts again. After each generation has completed, the remote shell performing the work automatically closes. The machine that was used is then considered available for the next piece of work that needs to be performed, as long as no-one has logged in to the machine in the meantime.

All the runs being executed use separate instances of the GP cycle as implemented by the manager script. This script uses an event-based mechanism to allow each run to continue asynchronously, preventing other runs getting held up if one piece of work is running on a particularly slow machine, for instance, or if a machine breaks down during execution. Synchronisation between jobs is only needed if individual-level parallelism is being used, where the results for the whole population need to be present before the selection process can take place.

Each level of control works by dividing the work up between languages according to their capabilities. The manager process is written in TCL (Tool Command Language, [Ousterhout, 1994]), which is very good for high-level work, co-ordinating the activity of a number of programs, and passing data between them. The TK toolkit is used to display results, get user input and enhance the interface to the system. C/C++ is good for numerical work and for storage of large data structures (such as genetic programs). A batch mode is included to allow overnight unsupervised processing of runs.

A deliberate decision was made to build GP-COM out of a large number of components, so that the system would remain flexible. This allows a new selection method, for example, to be added to the system simply by writing a short program that takes normalised fitness values as input and outputs a stream of commands in a form suitable for the population manager. This new method can then be specified purely by the
name of the executable file that implements the technique. No extra configuration, re-compilation or re-programming of the rest of the system is required. This allows the system to be changed and augmented quickly and with little effort.

A.3.2 The population manager
This component is the largest in GP-COM. Its role is to maintain and manipulate populations of genetic program trees. This involves permanent storage of the trees for logging purposes and volatile storage for manipulation. It provides facilities for creating a new population from an old, by duplicating populations, moving trees from one population to another and applying genetic operators to trees. The population manager is not specific to GP but is a general-purpose engine for creating and manipulating populations of genomes. The operations it provides are generic, the implementation of the genetic search cycle coming from a sequencing of commands to the engine provided by a high-level controlling script. The manager is also not tied to a specific function and terminal set — this information is provided at run-time.

This program is independent of the problem, and of the techniques selected for that problem. By implementing low-level operations, such as applying a genetic operator to an individual or duplicating a population, in the population manager, high-level functions (e.g. in GP, selection followed by population-wide application of genetic operators) can be implemented through structured application of these operations. Consequently the population manager need not incorporate selection methods directly, nor need it implement the genetic operators itself. These roles are taken by external programs which pass streams of commands to the population manager. A selection method, for example, need only know normalised fitness values and the population size. It is then able to generate commands to selectively copy members of an old population into a new, ‘blank’ one. Similarly, a genetic operator could take one or two individuals as input and output the offspring produced by the operator.

The population manager in GP-COM is capable of storing and manipulating arbitrary program trees. It provides checks for syntactic integrity according to information specified at run-time. It supports strongly typed primitives [Montana, 1995] and hierarchical ADFs (Automatically Defined Functions). It also provides support for using both together, i.e. it is possible to have strongly typed ADFs, but such a combination was not used in this thesis. Although this is a novel feature, it has not yet
been shown that the combination of these features is of value in practice, although both ADFs and strong typing have been shown to offer improvements when used in isolation ([Koza, 1994], [Haynes et al., 1995]). To retain the integrity of crossover and mutation operators with strong typing and ADFs, the following constraints are employed (see also section 2.6.3):

- Each individual in the population has an identical number of ADF trees. This ensures that any operation involving two individuals can make assumptions about those trees and their make-up.

- Each ADF must return a type that is already available and used. This is an inevitable consequence of the structure of an ADF, which is itself made up of zero or more ADFs and non-ADF primitives. The type of data returned by an ADF must therefore be based on the return type of a non-ADF primitive.

- An ADF tree of a particular index must have the same type and parameter list for all individuals in a population. So, if ADF1 for candidate 0 has return type double and takes (float,float,int) as parameters, ADF1 for all the candidates in a population must also have this prototype. This constraint guarantees correct crossover between candidates. The index of an ADF indicates which particular ADF is being referred to, i.e. in a population where individuals
have 3 ADFs, such functions would be called ADF0, ADF1, ADF2.

Crossover in a strongly-typed system incorporates the constraint that the sub-trees swapped between candidates must return the same type (or a member of the same equivalence class in the case of generic types). The operator used in GP-COM selects a tree at random from the first parent. It then searches the second parent for a sub-tree of identical type, starting from a random position in the genome and scanning the genome in both directions (the initial direction being randomly chosen) until such an element is found. If no such sub-tree is found in the entire genome of the second parent, a subtree of different type is selected from the first parent and the process repeats until a match is found. This operator is guaranteed to succeed because eventually a tree returning the same type as the whole program tree is selected — since all candidates return the same type there will definitely be a matching typed tree in the second parent, consisting of the entire genome of the second parent. This is a worst case scenario but it can still be of use, as although whole program trees may be swapped, any ADFs will not be.

There is a chance that the crossover operator will not choose the value-returning branch of the individual at all, but will choose to operate on an ADF. In this case, the third constraint above will force the operator to use the corresponding ADF of the second candidate as the other parent. This is necessary because each ADF has its own namespace, consisting of the formal parameters to that function and any other ADFs that may be callable from inside the selected ADF. Identical parameter names (such as p1, p2 etc.) in different ADFs may have different types, so crossover cannot ensure syntactic and type safety if crossing trees from ADFs with different indices.

Mutation for strongly-typed GP must produce a new sub-tree of the same type as the sub-tree it is replacing. Mutation of a strongly-typed ADF uses the primitive set of that ADF to ensure type safety.

A.3.3 Code Generation
The population manager component is independent of the problem domain, it cannot be used to execute the candidate programs it manipulates as it does not have access to the implementation of the primitives, only their prototypes. An external mechanism for evaluating program trees, completely separate from the population manager, is therefore required. One popular approach is to write an interpreter and execute the trees directly, but this involves considerable coding effort to produce a pseudo-language of sufficient
power, especially when supporting primitives of arbitrary type. These primitives would then have to be written in the pseudo-language also. A more powerful method is to use an existing language as the representation for program trees, thus taking advantage of that language’s capabilities in terms of storage and execution. Many systems use LISP as it provides a unified method of execution and of manipulating program trees and provides a lot of flexibility with little effort.

GP-COM originally took a novel approach to the problem of efficient execution of primitives, which also considered the other objective of minimising the amount of code to be written by the user. This method incorporates genetic material directly into the problem-specific program by translating the program trees to C source code and using machine-native binaries as the execution medium. This method is called the compilation approach.

Execution of candidate programs involves supplying a problem-specific code skeleton as a context into which the programs themselves can be inserted. GP-COM assumes that this skeleton is going to be a C or C++ program. Each primitive is written as a native-code function which simply returns its result. The population manager outputs each program tree as a (highly!) nested function call within a C function definition, matching exactly the structure of the tree, as well as an accompanying declaration. A similar process occurs for each ADF tree. The problem implementation is written in such a way that the existence of these program-tree functions is assumed, and this code is compiled in with the rest of the problem code for each generation. The functions that implement the genetic programs are replaced and re-compiled into the rest of the code each time a new population is produced.

The method reduces the amount of code the user and the system designer have to write. By adopting the in-built mechanisms for execution of code, and maintaining program and data stacks, in the compiler, the user is not troubled by these issues and can write primitives in a familiar language (e.g. C++). The system designer does not have to write an interpreter so the overall complexity of the system is kept down.

This compilation approach has important implications for performance of the system. A population of 500 individuals, each with 3 ADFs, produces a total of 2000 C functions each generation. These functions are added to the problem skeleton code and compiled to produce an executable which, when run, outputs the results of one generation of the run. Compilation of 2000 functions can take a very long time depending
on the equipment used, a penalty that is incurred each generation of every run. This system is however quite inappropriate for implementing steady-state genetic programming as compilation-time would cripple system performance if new individuals had to be added one at a time rather than in large groups.

In terms of performance, the compilation approach has benefits and drawbacks. Compared to systems that generate LISP functions and then evaluate those, a large compilation time can be a significant performance penalty. Typical compilation times for a population of 500 with a single ADF (making 1000 functions in all) on a Sun Sparcstation 5 ranged from 50 seconds to 6 minutes depending on the complexity of the program trees themselves. This may be compensated for by much reduced run-times from executing native binary code. The user also has the advantage of leveraging gains from optimisations present in whatever compiler is being used, possibly increasing run-time performance even further. Overall performance gains or losses are thus determined by the execution cost of the problem and the size of the population. A computationally expensive problem could gain considerably from being coded in C rather than LISP, and this may more than compensate for the compilation overhead. Additionally, compilation time is roughly linear in the size of the population, so growth in computational resources overall remains linear. This means that the compilation time can be viewed as a constant factor in the total execution time of one cycle of the genetic search process. Problems which are inherently expensive to run, even with the speed advantages of C, are thus affected to a lesser degree than trivial problems which execute in a few seconds. Performance gains could also be made by incorporating a system similar to the UNIX ‘make’ command, where only candidates which have changed would need to be recompiled with the next generation.

Where compilation does score favourably is in the flexibility of the function and terminal sets. This allows the user to incorporate functionality from a huge range of software. Data types used in strongly-typed GP can map directly to types defined in C++, including of course user-defined classes. Functions and terminals in the primitive set map directly to C++ functions and variables and as such can be drawn from any library and linked in to the problem skeleton. This allows the program trees to correspond very closely to the domain representation as this is itself a model using the same set of primitives. This powerful representational advantage has an added bonus, that the user need write no code other than the primitives to get the genetic programs to execute. Run-
time code is not needed since it is taken care of by the compiler automatically.

This method was eventually replaced by a more traditional execution method based round pseudo-interpretation. The reason for this was that during experiments, runtimes were often not the significant factor they were previously thought to be. Most effort and time was spent in exploration, looking for suitable fitness functions and primitive sets, so it was more important to be able to try out ideas quickly than to spend long periods on a single run with extremely long runtimes. Compilation time thus becomes the dominating factor in run execution times.

This second method of program execution is more traditional but retains much of the performance advantage that can come from the compilation method. A standard set of routines is provided for the user to permit reading and writing of genomes to and from disk, along with code to read and write configuration details for particular runs (see section A.3.7). These routines are implemented as C++ code. Further libraries implement a tree-based execution system designed to allow the user to execute genetic programs from data read in at run-time. This is an extremely lightweight interpreter that does none of the tasks usually required by a normal interpreter such as type checking and syntax checking, since this should be assured by the population manager which guarantees correct genomes will be produced. Genomes are read in at run time and each element of the genome is mapped by a user-customisable function to a C++ function known about at compile time. This mapping is specified by the user, and each function to which there is a mapping implements a single primitive in C++ (or other linkable language) code. Information about the arity of each primitive is read in from the same configuration information as is used by the population engine, and used to construct a tree of nodes, each of which contains a pointer to the primitive function to be executed at that node.

The primitives themselves deal with the necessary passing of data back up the program trees through a series of global stacks. One stack is instantiated (from a templated stack class) by the user for each type of data used in the primitive set. Each primitive simply pops its arguments from the stack of the appropriate type, and pushes the result of its computation onto the stack of the result's type. The execution order is specified by the tree construction process which always implements traditional depth-first execution.

This execution method combines the good features of both the interpreted and the compiling method. The flexibility of the interpreter is present, so the problem-specific
executable can read and write genomes without re-compiling. The execution speed is almost identical to the compilation method with a small overhead for the stack manipulations. This method is semi-interpreted, the primitives themselves are compiled but the order in which they are called is determined at run-time. It gains a speed advantage from the compilation method and has the extra flexibility associated with interpretation.

Divorcing execution of genetic programs from their manipulations can be advantageous since it allows the user to take advantage of special characteristics of the execution medium which can help speed up the process. In the experiments performed for this thesis this means the speed of native C++ code helps cut down execution times. Other features which could be employed by different execution environments could be utilisation of high-quality code libraries for particularly complex or specialised problems, or added support for parallelisation to increase execution speed or even as an inherent part of the problem domain. Such enhanced features are much easier to employ if the GP code is kept as far as possible from the execution of the programs as the choice of execution modes is much greater. This allows the program code to run in environments totally suited to its nature and fewer compromises have to be made in implementation.

A.3.4 Parallelism in Genetic Programming

[Koza, 1992] gives three levels at which the performance of GP may be increased by parallelisation. These are:

- By fitness case. Every individual in a population is evaluated on every processor, but only a subset of the fitness cases are evaluated on each processor.
- By sub-population. Every fitness case is evaluated on every processor, but only for a distinct subset of the population.
- By run. A problem will most likely require several runs to produce an adequate solution. These runs can be simultaneous, with each processor evaluating the whole population (for one run) on every fitness case.

GP-COM has built-in support for parallelisation by sub-population and by run. Run-level parallelism simply uses a separate machine for each run that is active. To parallelise by sub-population, GP-COM can run a single generation on multiple machines
and assign a separate portion of the population to each machine. This is a simple process that relies on a very simple interface being supported by the problem-specific code. Parameters are passed to the executable indicating which section of the population that instance is to execute. Partial results are written out and centrally collated by the run manager. Currently GP-COM supports up to 5-way parallelisation by sub-population, giving tremendous speed increases. The two methods for parallelising GP in GP-COM are illustrated in Figure A.3.

Work is automatically launched on idle machines to ensure maximum performance, looking first for faster machines but only ever running on machines on which no-one is logged in. This is to minimise the disruption caused by the experiments to other users, a necessary precaution when utilising idle CPUs on machines that belong to other people! A centralised manager process farms out work to the various machines on a generational basis. Each job executes for a duration of one generation, whether on a whole population or a section of a population, writes out its results (fitness values) and returns. The manager process then farms the work out afresh once the population
The algorithm for farming out work to machines ensures that faster machines are used whenever possible. Each machine that is being used is removed from a global list of available machines while it is executing a generations-worth of work. Once completed, the machine name is added to the front of the list which ensures it will get picked first for any future work. Faster machines will stay near the front of the list since they will finish jobs more often than slower machines. This simple greedy algorithm ensures that the best use is made of available computing resources throughout the session.

The manager can support multiple asynchronous runs simultaneously and both run-level and individual-level parallelisation within a single session. The amount of work that can be done simultaneously is limited by the speed of the central workstation that runs the population manager, which has to queue operations for each executing run and process them in turn. A faster machine would allow more machines to be used in tandem.

The location of the bottleneck in the system depends on the runtimes incurred in fitness evaluations: a long runtime means the central processor has more time to perform population manipulations and may be idle for some of the period, shorter runtimes mean the central workstation may become a bottleneck. In practice, another limitation is the number of free machines available at any one time, especially during the day. As an example, the template matching problem in Chapters 5 and 6 took about 6 seconds per generation on the faster machines available. For this problem ten parallel runs were sustainable before the central workstation (a Sun Sparcstation 5 with 96MB of RAM) became the bottleneck.

GP-COM supports multiple architectures and is currently used on a heterogeneous network of Solaris and SGI machines. This support is in part gained by the use of portable, high-level commands and languages to organise the distribution of processing. The location of components during a run can be seen in Figure A.4.

A.3.5 Tree Generation
Program trees in GP-COM can be of arbitrary size and complexity. Instead of imposing a fixed depth limit on tree generation, GP-COM uses a probabilistic method to control the size of trees generated. This method uses a terminal-probability, a floating point value under user control, dictating the likelihood of choosing a terminal at each node during tree generation. This figure is a run-specific parameter which applies at the root node of
the tree. This probability is increased additively with the depth of the current node, until at some depth it reaches 1. This increase for a given subtree of a node is multiplied by the branching factor of that node, so a function taking 5 parameters is less likely to have large sub-trees underneath it that one taking 2 parameters at the same depth.

Additionally, a separate `mutate-terminal-probability` is used to control the generation of sub-trees by mutation. This controls the way in which trees grow as the run progresses. Given that the crossover operator does not change the total size of its operands, and assuming that the genetic process does not have some in-built or fitness-inspired bias toward small trees, the mutation operator is responsible for any changes in the average size of candidate trees. Tuning of these probabilities allows fine control over the initial size and growth of program trees. These two probabilities are also implemented independently for ADF tree creation and mutation.

This method avoids the use of hard depth limits for trees which could (depending on the problem definition) preclude the formation of certain solution forms. It is particularly suited to trees using the chaining structuring method (see Chapter 6) which tend to be extremely deep without necessarily having high node-counts. An additional advantage is that it reduces the number of parameters to be tuned by the user: the probabilistic method requires two parameters to be set, `terminal-probability` and `mutate-terminal-probability`. The traditional methods as used in [Koza, 1992] require the user to
set maximum depth limit, minimum depth limit, ratio of full to grow methods (i.e. the choice of method - this could be minimised to a single parameter with, say, a value of 0 being ‘use only the full method’, 1 being ‘use only the grow method’, 0.5 indicating a half-and-half ramp), and a mutation tree depth limit. If mutation is not being used, both approaches to generation use one less parameter.

A.3.6 Storage of genetic material
GP-COM stores genomes as linear structures which, when combined with prototype information for the primitives, can be mapped to tree structures and back again. This representation is human-readable and easy to parse, which aids debugging and analysis of populations, and the writing of custom applications for problem-specific reporting. For instance, a program tree that codes a function for a problem such as symbolic regression, can easily be converted to a form suitable for visualisation by a plotting program. As an example, GP-COM includes a utility to view program trees by converting these representations to a form readable by a graph layout tool (see Figure 4.8 for a genome that has been output in this way).

A.3.7 Run-time customisation
GP-COM is a hierarchical system and has a default global configuration which is inherited each time a new problem is created. These configurations are used by many of the components in the system and are easily edited by the user. In turn, these problem-specific configurations are automatically inherited by each run, allowing the user to set options that apply to all runs for a given problem, or to override them for a particular run or set of runs. Options currently implemented are:

- mutation rate and crossover rate
- terminal-probability and mutate-terminal-probability
- ADF-terminal-probability and ADF-mutate-terminal-probability
- Population size
- Selection method. These are implemented as separate programs and take inputs from normalised fitness values. They output a stream of commands which are piped into the population manager. These commands consist of instructions to place a particular member of the current population into a slot in the next population.
• Primitive set. This is a pointer to a *types file* containing the prototypes for a given set of primitives. This allows rapid changing of primitive sets with no recompilation, as each primitive set can be a subset of those functions already defined in C code. This allows quick experimentation with different primitive sets, for example to examine the effect of adding or removing a primitive from the set. Type names are irrelevant to their representations in the underlying implementation, and may include pointers, allowing the manipulation of type systems (as seen in Chapter 6) without necessarily requiring a re-writing of the problem-specific programs that execute the candidate solutions.

• Number of ADFs. If set to 0 no ADFs are used. If non-zero, the system will generate prototypes randomly for each ADF at generation 0 and add them to the types file. This allows each run to be different in terms of the prototypes of its ADFs. If the types file contains prototypes for ADFs already then these details are used, so runs can be forced to use the same ADF prototypes by adding them to the types file at the problem-level.

• Random seeds. These are generated using an architecture-independent (Park-Miller [Park & Miller, 1988]) generator. This allows the system as a whole to be repeatable - a run started using the same configuration and same random seeds will produce the same results every time even if run on different architecture machines (a necessary property if repeatability is to be maintained within a parallel execution environment).

• The number of generations to perform. In the absence of a specific piece of code (supplied by the user, in the file structure of the problem itself) that performs the termination criterion, a maximum number of generations is assumed to be the limit to the run’s progress.

• Method of parallelisation – run-level or population-level.

### A.4 Discussion

The design of GP system software can affect the ability of the researcher or practitioner to function effectively and can help or hinder the use of GP when applying the technique.
to new problem domains. Often, reasons for success or failure in a GP run are hard to find due to a lack of information, or the tools to extract such information.

Although GP-COM is not intended as the last word in GP software design, it serves to highlight many of the important issues that must be considered when preparing to use GP in a series of experiments. The system was created because none of the systems available at the time supported such enhancements as strong typing, which it was anticipated would be heavily used. Once the need had become apparent for a new system to be created, other factors such as flexibility and performance had to be considered. Although continually developed for the experiments in this thesis, GP-COM is now three years old and it is not intended to release the system for general use. GP-COM is presented here as a case study to bring to the fore software engineering issues that can affect the ability of the researcher to design and implement experiments with GP.

Flexibility and performance must be considered when choosing or creating GP software. A design has been presented that closely maps software components to components in the conceptual description of GP, integrating the user's conceptual model with the implementation framework. This component-based architecture permits great flexibility in configuration with minimum effort, allows the most appropriate tools and languages to be used where the benefit is greatest, and promotes high performance through parallelism and the use of heterogeneous architectures.